



GPA-789

Analyse et conception orientées objet

Notes de cours

GPA-789 ANALYSE ET CONCEPTION ORIENTÉES OBJET

Notes de cours

Auteur: Tony Wong, Ph.D., ing.
Département de génie de la production automatisée
École de technologie supérieure
courriel: tony.wong@etsmtl.ca

Ce document a été rédigé pour les étudiants
du cours GPA789 — Analyse et conception orientées objet.
Tous les produits commerciaux mentionnés dans ce document
sont des marques déposées de leurs propriétaires respectifs.
Le genre masculin est utilisé dans ce document afin d'alléger la lecture du texte.

Table des matières

CHAPITRE 6

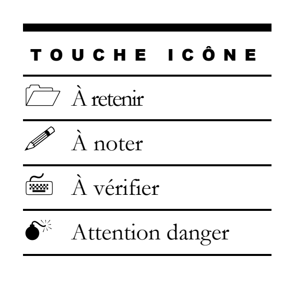
CHAPITRE 7

CHAPITRE 8

Liste des figures

Liste des tableaux

CONVENTION TYPOGRAPHIQUE



La lecture de ce document est nécessaire pour tous les étudiants du cours GPA789 Analyse et conception orientées objet. Afin de faciliter le repérage des éléments importants contenus dans ce document, un système de notation iconique est utilisé. Ces icônes de rappel sont montrés dans la colonne à gauche. L'icône signifie que l'information est essentielle à la bonne compréhension de la présentation et peut être invoquée dans les sections subséquentes de ce document. L'icône signifie que les données présentées peuvent être utiles dans des procédures et des démarches décrites. L'icône dénote une procédure ou manipulation à tenter ou à essayer. Enfin, l'icône indique un problème potentiel qu'il faut éviter.

Outre le système iconique, une convention typographique particulière est également adoptée pour aider la lecture de ce document. Le tableau 1 présente l'usage de cette convention.

Type de caractères	Contexte d'utilisation
cmd1	Les commandes à donner ou le corps d'un code source.
DMA (*Direct Memory Access*)	Les acronymes sont toujours explicités entre parenthèses. Si l'acronyme est d'origine anglaise, les mots entre parenthèses seront imprimés en caractères italiques.
Quelques « saveurs » d'UNIX	Les mots placés entre guillemets (français) doivent être pris dans leurs sens figuratifs. Les guillemets sont également utilisés pour désigner les ternes techniques ou les options d'un menu.
Le **remplacement** des programmes	Les mots imprimés en caractères gras représentent un concept ou simplement pour mettre de l'emphase sur les mots. Le concept en question est normalement expliqué plus loin dans le texte.
Le micro-noyau d'UNIX |	Le texte en retrait accompagné d'un trait vertical désigne un commentaire personnel de l'auteur. Les commentaires sont toujours situés dans l'espace libre entre la marge de gauche et le corps du texte.

Tableau 1 Convention typographique adoptée.

Enfin, pour la présentation des instructions et programmes informatiques, le code source sera entouré d'un cadre de la manière suivante :

```
// boucle infinie
for (;;) ;
while (1) ;
```

■

NOTES AUX ÉTUDIANTS

L'objectif premier de ce document est de présenter le contenu du cours GPA789 d'une manière claire, précise et agréable à lire. Le format de présentation adopté ressemble à celui d'un manuel de référence. L'espace entre la marge de gauche et le corps du texte est réservé pour l'inscription des notes personnelles. Chaque chapitre traite un sujet distinct. Les sections et sous-sections des chapitres sont utilisées pour présenter logiquement le contenu de la matière concernée.

Deux sections spéciales sont incluses à la fin de chaque chapitre. La section intitulée « Lecture suggérée » donne les références supplémentaires qui aideront à mieux assimiler la matière présentée. La section intitulée « Problèmes » propose une série d'exercices qui aideront à évaluer le niveau de compréhension du lecteur.

Les informations disponibles dans l'Internet sont privilégiées. La raison principale de cette décision est justement la disponibilité de ces informations. Il est possible de les obtenir en tout temps pourvu que l'on dispose d'un ordinateur relié à l'Internet. Cependant, l'Internet en tant que bibliothèque virtuelle possède une lacune majeure : la volatilité des sites qui entreposent ces informations. Le va-et-vient des sites fait en sorte que la permanence des informations n'est plus assurée. Une solution mitoyenne consiste à ne citer que des sites *réputés*. Autrement dit, les sites universitaires et les sites de grandes corporations seront privilégiés au détriment des pages HTML personnelles.

Le degré de difficulté des problèmes proposés est indiqué par un système de symboles. Les problèmes notés par le symbole * sont les plus simples à résoudre tandis que ceux notés par les symboles **** sont les plus difficiles et demandent probablement un temps plus longs à les solutionner. De plus, plusieurs problèmes nécessitent une certaine connaissance dans le domaine de la programmation informatique. Ces problèmes seront notés par le symbole ⌨. Ceux qui sont faciles à réaliser sont notés par ⌨. Pour les problèmes exigeant une programmation plus ardue, ils sont notés par ⌨⌨⌨⌨.

L'une des difficultés rencontrées dans la rédaction de ce manuel est l'utilisation correcte des termes techniques. Le domaine informatique est truffé d'anglicisme et d'acronymes. L'auteur a adopté, dans la mesure du possible, les recommandations contenues dans « Le grand dictionnaire terminologique » de L'Office de la langue française du Québec afin de donner un sens juste et précis aux mots techniques qui parsèment ce document. Cependant, certaines équivalences anglaises – françaises ont une portée trop générale. Dans de tels cas, l'auteur privilégie le sens américain (et non l'anglicisme) plutôt que celui donné dans le dictionnaire terminologique. Aussi, pour aider à la compréhension, les termes américains sont souvent placés entre parenthèses en caractères italiques.

Enfin, malgré toutes les précautions mises en place lors de la rédaction, des erreurs peuvent encore se trouver parmi les pages de ce manuel. Veuillez communiquer avec l'auteur de ce manuel afin de rapporter les erreurs rencontrées. Les commentaires qui peuvent aider à l'amélioration de ce manuel sont également les bienvenus. L'adresse courriel de l'auteur est indiquée dans la page de présentation précédant la table des matières.

Tony Wong
Montréal, janvier 2001

CHAPITRE

1

Éléments du langage C++

Un système de programmation, $\{\phi_i / i \in N\}$, est une énumération de toutes les fonctions partielles récursives, c'est un autre synonyme pour une numérotation de Gödel.

— *Bernard M. Moret, The theory of Computation.*

Dans le domaine de la programmation orientée objet, le langage C++ demeure l'un des langages les plus populaires. Il est fort utilisé parce qu'il donne une grande liberté aux programmeurs. Il demeure populaire parce qu'il est un langage de programmation mature. Nous présentons dans ce chapitre les éléments importants de C++. Le but est de faire comprendre son utilisation dans les méthodes de conception orientée objet. Enfin, les différentes constructions montrées dans ce document peuvent être utilisées dans la phase de réalisation orientée objet d'un programme informatique.

1. ÉLÉMENTS DU LANGAGE

L'unité de base du langage C++ est appelée jeton[1]. Les jetons sont composés d'un ensemble de caractères alphanumériques. Ils constituent le vocabulaire de base du langage et certains sont réservés dans leur usage pour donner un contexte au compilateur. La sémantique d'une énoncée en C++ est donc déterminée par l'utilisation de ces jetons. Enfin, les jetons sont séparés par des blancs (des espaces ou des caractères de tabulation).

Il existe dans le C++, cinq (5) types de jetons : *i*) mot clé; *ii*) identificateur; *iii*) littéral; *iv*) ponctuation; *v*) opérateur. De plus, deux types de jetons spéciaux désignent le texte des commentaires.

1.1 MOTS CLÉS

Les mots clés sont des jetons réservés et possède une signification prédéterminée pour le compilateur. Ces mots clés sont énumérés dans le Tableau 2.

[1] Tous les langages de programmation sont composés de jetons.

Jeton	Signification
and	ET logique. Identique à l'opérateur &&.
and_eq	Assignation-ET des bits. Identique à l'opérateur &=.
asm	Insertion des directives en langage assemblé.
auto	Déclaration d'une variable locale.[2]
bitand	ET appliqué à des bits. Identique à l'opérateur &.
bitor	OU appliqué à des bits. Identique à l'opérateur |.
bool	Type booléan et accepte comme valeur : true, false.
break	Arrêt de l'exécution d'une boucle.
case	Partie de switch-case donne le choix du chemin d'exécution en fonction de la valeur retournée par switch.
catch	Élément de la structure de gestion d'exception. Indique le type d'exception à gérer.
char	contient 1 octet de donnée.
class	Déclare l'existence d'une classe ou définie un objet d'une classe.
compl	Complément logique. Identique à l'opérateur ~.
const	Déclare une constante. Il est préféré à l'usage de #define (vestige du langage C).
const_cast	Permet la conversion d'un objet de type const en un autre type.
continue	Permet le passage immédiate à l'itération suivante dans une boucle.
default	Exécution d'un choix par défaut dans la construction switch-case.
delete	Élimine un bloc de mémoire alloué par new.
do	Boucle do-while.
double	Type nombre point flottant (8 octects).
dynamic_cast	Conversion d'un type d'objet en un autre type.
else	Construction if-else.
enum	Spécifie un type énuméré.
extern	Déclare un objet qui n'est pas visible dans le fichier en cours.
float	Type nombre point flottant (4 octects).
for	Élément d'une boucle.
friend	Permet à une fonction d'accéder aux méthodes et données protégées (ou privées) d'une classe.
goto	Hé oui. Il existe dans le C++.
if	Construction if-else.
inline	Indique au compilateur de placer le bloc de code directement à l'endroit de son appel.
int	Type nombre entier (grandeur équivante à un mot machine. Donc dépendant de l'architecture de l'ordinateur).
long	Type nombre enttier (4 octects).

[2] Toujours assigné par le compilateur pour des variables déclarées à l'intérieur d'un bloc.

`namespace`	Permet la création d'un nouveau espace de noms.
`new`	Création dans l'espace mémoire d'un objet (allocation dynamique de la mémoire).
`not`	Inversion logique. Indentique à l'opérateur `!`.
`not_eq`	NON-ÉGAL logique. Identique à l'opérateur `!=`.
`operator`	Permet la définition d'un opérateur associé à une classe.
`or`	OU logique. Identique à l'opérateur `||`.
`or_eq`	OU-ÉGAL logique. Identique à l'opérateur `|=`
`private`	Niveau d'accès privé. Seulement les fonctions membres et amis (`friend`) d'une même classe.
`protected`	Niveau d'accès protégé. Seulement les fonctions membres et amis (`friend`) d'une même classe et ses classes dérivées.
`public`	Niveau d'accès public. Les fonctions membres et les données membres déclarées ainsi sont accessibles à tous.
`register`	Indique au comiplateur que l'on désire placer une variable ou une adresse de fonction dans un des registres du processeur. Le compilateur peut ne pas respecter ce souhait.
`reinterpret_cast`	Permet la conversion de n'importe quel type d'objet en n'importe quel autre type d'objet. Permet la conversion d'une variable de type intégral en un pointeur et vice versa.
`return`	Provoque la fin d'une fonction.
`short`	Type nombre entier. (2 octects).
`signed`	Équivant à `int`.
`sizeof`	Donne la grandeur en octects d'un type.
`static`	Modificateur donnant une durée de vie à une variable identique à celle du programme. Appliqué devant une fonction, le modificateur restreint la visibilité de la fonction à l'intérieur du fichier source.
`static_cast`	Conversion d'un type en un autre type. Aucune vérification n'est réalisée à l'exécution du programme.
`struct`	Identique à une classe (`class`) mais toutes les fonctions membres et les variables sont de niveau d'accès public.
`switch`	Construction `switch-case`.
`template`	Mot clé pour créer un type paramétrisé.
`this`	Représente le pointeur de l'objet lui-même.
`throw`	Élément de la structure de gestion d'exception. Provoque la création d'un objet représentant l'exception générée.
`try`	Élément de la structure de gestion d'exception. Surveille une section de code pour la possibilité des exceptions.
`typedef`	Définir un nom synonyme pour un type donné.
`typeid`	Permet la détermination d'un type pendant l'exécution du programme.
`union`	Permet la création d'une structure pouvant contenir des types différents à différents moments.
`unsigned`	Équivalent à `int` mais non signé (nombres positifs).
`virtual`	Déclare une fonction ou une classe virtuelle.
`void`	Type nul.

`volatile`	Indique au compilateur de ne pas appliquer les algorithmes d'optimisation sur une variable. Une variable est modifiée par `volatile` si elle est susceptible d'être modifiée par le système d'exploitation ou matériels périphériques.
`while`	Mot clé d'une boucle `do-while` ou `while { }`.

Tableau 2 Jetons du langage C++.

1.2 IDENTIFICATEURS

Les identificateurs sont des séquences de caractères alphanumériques ne débutant pas par un chiffre. Le compilateur utilise, d'une manière interne, le caractère souligné (_) comme premier caractère des identificateurs lors de la traduction du code source en code intermédiaire. La convention veut que le nom des constantes, des macros (directives pour le préprocesseur) et les structures (`struct`) soit écrit en majuscule. Les identificateurs servent à identifier les noms de variables, fonctions et classes.

1.3 LITTÉRAUX

On appelle les valeurs constantes des littéraux. Tous les types de base de C++ peuvent contenir des littéraux. Par exemple `3.141516`, `5`, `2.1819e-1` sont des littéraux.

1.4 PONCTUATION

La ponctuation utilisée dans C++ est plutôt simple. La liste des ponctuations possibles comprend : *i*) les accolades (`{}`); *ii*) les virgules (`,`); *iii*) les deux-points (`:`); *iv*) les parenthèses; *v*) les points-virgules (`;`); *vi*) les guillemets et double guillemets (`'` et `"`); *vii*) les crochets (`[]`). La ponctuation confère au langage C++ une structure syntaxique appropriée.

1.5 OPÉRATEURS

On désigne un opérateur par une séquence contenant un (1) ou plusieurs caractères. Il permet la réalisation d'opérations arithmétiques ou logiques dans une expression. De plus, les opérateurs définis par le programmeur peuvent élargir la signification des opérateurs de base. Par exemple, on peut définir l'opérateur `+` comme l'insertion d'un objet dans une liste. Le Tableau 3 donne la liste des opérateurs de base du langage C++.

Opérateur	Signification
`%`	Reste d'une division. Ne s'applique qu'à des entiers.
`+`	Addition ou positif unaire (ex : `+5`).
`++`	Pré (Post) Incérmentation.
`-`	Soustraction ou négatif unaire (ex : `-5`).

--	Pré (Post) Décrémentation.
*	Multiplication ou déférence (ex : `*x = 5`).
/	Division.
!	NON logique.
<	Inférieur à.
<=	Inférieur ou égal à.
>	Supérieur à.
>=	Supérieur ou égal à.
==	Égalité.
!=	Inégalité.
&&	ET logique.
||	OU logique.
&	ET bit à bit.
|	OU inclusif bit à bit.
^	OU exclusif bit à bit.
?	Conditionnel.
<<	Décalage des bits vers la gauche ou insertion dans un flux de sortie.
>>	Décalage des bits vers la droite ou extraction à partie d'un flux d'entrée..

Tableau 3 Opérateurs de base du langage C++.

1.6 TYPE DE DONNÉES DE BASE

Le C++ dispose d'un nombre de types prédéfinis que l'on peut utiliser directement dans nos programmes. Ces types de données sont énumérés dans le Tableau 4.

Type	Étendue
Donnée	entier (`int`, `short`, `long`) point flottant (`float`, `double`, `long double`) caractère (`char`) chaîne de caractères (`char*`) booléen (`bool`)
Variable constante	Constante entière littérale (décimale, octale, hexadécimale) Signée/non signé et longues (8L, 8LU, 8lu) Constante point flottante (notation scientifique ou décimale) Simple précision et double précision (3.1416e-3F, 3.1416e-3L) Constante caractère littéral (`'a'`, `'1'`) Caractère non imprimable (`\n`, `\?`) Constante chaîne littérale (`"a"`, `"Allô"`)

Variable symbolique	Les variables symboliques possèdent une valeur et une adresse mémoire 1. int x = 10; 2. int y = x + 5; La ligne 1. définie une variable entière `x` et l'initialise à la valeur 10. La mémoire pour entreposer la valeur de `x` est également allouée. La deuxième ligne effectue une addition de x avec la constante littérale 5. Le résultat est placé dans la variable `y`. Noter que `x` est appelée *lvalue* dans la ligne 1. parce qu'elle reçoit une valeur.
Variable pointeur	La valeur contenue dans une variable pointeur est une adresse mémoire. On peut donc référencer indirectement une autre variable à l'aide d'une variable pointeur. 1. int j = 500; 2. int *pj = &j ; 3. int k = 2 * (*pj); La variable pointeur `pj` *pointe* à l'adresse mémoire de `j`. La notation `*pj` signifie le contenu de la variable pointée par `pj`. Dans ce cas, `pj` pointe à la variable `j`, donc `*pj` égal à 500. La valeur de `k` à la ligne 3. est donc 1000. Par définition, C++ traite les chaînes à l'aide de pointeurs de caractère (`char*`). Ainsi, Char *pstr = "Allô" est une chaîne de caractères pointé par `pstr`. Noter que les chaînes (et les tableaux) natives de C++ sont toujours terminées par la valeur zéro (0). Ainsi, nous avons `pstr[0] = 'A'`, `pstr[1] = 'l'`, `pstr[2] = 'l'`, `pstr[3] = 'ô'` et `pstr[4] = 0`.
Constante	Le mot clé `const` modifie une variable en une constante symbolique. Une constante symbolique doit toujours être initialisée. Const float PI = 3.1416; Const unsigned int CERCLE = 101U; Une variable déclarée constante ne peut être modifiée directement par le programme. On peut cependant affecter une constante symbolique à un pointeur modifié par `const`. Const int *pPI = &PI; Puisqu'une constante occupe une adresse mémoire.
Référence	Une variable référence est un alias d'une autre variable. Ainsi, 1. double x = 12345.678; 2. double &refx = x; 3. Refx *= x; La variable référence `refx` est identifiée par le symbole `&` et elle fait référence à la variable `x`. La ligne 3. effectue la multiplication du contenu de `x` par le contenu de `x`. Ne pas confondre: 1. double *px = &x; 2. double &refx2 = x; Dans la ligne 1. `&x` signifie l'adresse de `x`. Alors que dans la ligne 2. `refx2` est une variable de référence et est un alias de `x`. Une variable référence est toujours précédée de son type. Enfin, on dit que `&x` de la ligne 1. est un *rvalue* que l'on affecte à `px`. De même `x` est un rvalue de `refx2` dans la ligne 2.

Énumération	L'énumération permet la détermination d'une plage de valeurs intégrales. Par exemple l'énumération Enum COULEUR {NOIR, ROUGE, VERT, BLEU}; Définie une plage de valeurs croissantes pour le type `COULEUR` représentées par les symboles `NOIR`, `ROUGE`, `VERT` et `BLEU`. On peut donc remplacer des constantes numériques par des étiquettes symboliques. Son utilisation rend la lecture du code source plus facile. COULEUR macouleur = ROUGE; est beaucoup plus compréhensible que unsigned int macouleur = 101U; À noter que la définition de l'énumération n'alloue pas d'espace mémoire. On ne peut pas appliquer une variable pointeur (ou référence) à une énumération.

Tableau	On définit un tableau par la syntaxe suivante:
	1. float ftab[450];
	2. int itab[10][12];
	Dans la ligne 1, un tableau unidimensionnel de 450 éléments de type `float` est créé. Dans la ligne 2, un tableau bidimensionnel de 10 par 12 est créé et il doit contenir des entiers. La dimension des tableaux doit être connue lors de la compilation. Si la dimension n'est pas connue d'avance, il faut utiliser l'allocation dynamique de la mémoire pour la création des tableaux. On peut assigner un élément du tableau par:
	int y = itab[1][2];
	ftab[5] = 3.1416;
	Donc, un tableau peut jouer le rôle de *lvalue* et *rvalue*.
	La relation entre un pointeur et un tableau est intimement liée. Par exemple,
	float x = ftab[5];
	float y = *(ftab+5);
	donne la même valeur dans `x` et dans `y`.

typedef	Ce mot clé permet d'assigner à un identificateur un type et l'identification devient synonyme du type dans son utilisation. Par exemple, typedef unsigned int UINT; rend l'identificateur `UINT` synonyme du type `unsigned int`. Ainsi, unsigned int x; UINT y; déclarent `x` et `y` des variables de type entier positif.

Tableau 4 Types de données de base en C++.

Nous verrons plus loin la nature exacte et le contexte d'utilisation de ces différents types de données.

1.7 INSTRUCTIONS

Une instruction, une fois compilée, est une unité exécutable d'un programme C++. Toutes les instructions sont terminées par un point-virgule. Il est parfaitement légal d'avoir une instruction qui ne contient qu'un point-virgule (instruction nulle) et elle est souvent utilisée dans la programmation de bas niveau[3]. Tous les exemples dans le Tableau 4 représentent des instructions légales en C++.

[3] Pour générer des instructions no-op en langage assemblé.

1.8 INSTRUCTIONS COMPOSÉES

Une instruction composée est représentée par une séquence d'instructions entre les accolades { et }. Les instructions composées servent à regrouper une séquence d'instructions en un seul ensemble consistant et logique. On peut définir des variables locales à l'intérieur des instructions composées. La portée de ces variables locales ne dépasse pas l'étendue délimitée par les accolades.

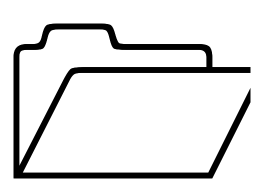
1.9 EXPRESSIONS

Les expressions dans le C++ sont composées d'opérations. On représente les opérations par l'utilisation des opérateurs unaires (une seule opérande) et binaires (deux opérandes). L'évaluation des expressions est réalisée par le compilateur (si le contexte le permet) ou par le code exécutable lors de l'exécution du programme. L'ordre de l'évaluation est déterminé par les règles arithmétiques et l'associativité des opérateurs. Certains opérateurs peuvent prendre plusieurs significations dépendant du contexte dans lequel ils sont utilisés. Par exemple,

```
int x = 1234;
int *y = &x;
float z = 3.1416 * (*y);
```

L'opérateur multiplication (*) utilise le même symbole que l'opérateur d'indirection. Dans ce cas, il est préférable de mettre entre parenthèses `*y` pour éviter toute confusion possible[4].

1.10 DÉCISION ET CONTRÔLE DE L'EXÉCUTION



L'exécution d'un programme informatique est souvent dictée par des conditions déterminées en-ligne. Pour pouvoir diriger le flux, le langage C++ offre un ensemble d'instructions pour le contrôle logique de l'exécution du programme. La condition de bouclage, c'est-à-dire, l'expression qui indique le nombre ou la limite des itérations peut avoir une valeur numérique quelconque (entière, simple précision, double précision) ou avoir une valeur booléenne quelconque. L'importance n'est pas la valeur elle-même mais bien si l'expression évalue à zéro (0) ou non. La condition de bouclage est fausse si l'expression évalue à zéro. Elle est vraie si l'expression est non nulle.

> Fait attention à des conditions de bouclage impliquant un résultat en point flottant. La perturbation introduite par la troncature des nombres peut donner une condition de bouclage erronée !

[4] Il est parfaitement correct d'écrire `float z = 3.1416 * *y;` mais beaucoup moins lisible.

1.10.1 INSTRUCTION IF

L'instruction `if` permet la vérification d'une condition particulière. La syntaxe de l'instruction `if` est :

```
if (expr) { instr; }
```

où `expr` est une expression et `instr` est une ou plusieurs instructions (instruction composée) à exécuter si `expr` est évaluée à une valeur non nulle. Dans le cas contraire `instr` ne sera pas exécutée. Pour donner une plus grande souplesse à l'instruction `if`, on peut ajouter une clause alternative par l'utilisation de l'instruction `else`. Ainsi,

```
if (expr) { instr1; }
else { instr2; }
```

L'instruction (ou l'instruction composée) `instr1` est exécutée si la valeur de `expr` est non nulle. Autrement, le contrôle est passé à `instr2`. Enfin, on peut aussi réaliser des tests multiples à l'aide de `if-else`.

```
if (expr1) { instr1; }
else if (expr2) { instr2; }
else if (expr3) { instr3; }
else { instr4;}
```

Dans cette dernière construction, `instr1` est exécutée si `expr1` est non nulle. Autrement `instr2` est exécutée si `expr2` s'avère non nulle et ainsi de suite. Si toutes les expressions sont évaluées à nulle alors `instr4` est exécutée. La dernière instruction `else` est facultative car elle réalise une exécution par défaut de l'instruction `if`. À noter que la construction `if-else` n'admet qu'une seule exécution de l'ensemble des conditions. Autrement dit, si `expr2` et `expr3` sont toutes deux non nulles, seule l'instruction (ou instruction composée) `instr2` sera exécutée puisqu'elle précède celle de `instr3`.

1.10.2 CONSTRUCTION SWITCH-CASE

Dans le cas où l'on posséderait des sections de code à exécuter et qu'elles sont dépendantes d'un ensemble de valeurs, il serait préférable d'utiliser la construction `swtich-case`.

```
switch (6000 - id) {
  case 101 : { instr1; break; }
  case 102 : { instr2; break; }
  case 103 : { instr3; }
  case 104 : { instr4; break; }
  default : { instr5; }
}
```

Dans l'exemple ci-dessus, la valeur donnée par l'expression accompagnant `swtich` est de type intégral (c'est-à-dire, nombre entier, caractère, etc.). La valeur constante des `case` indique le cas à traiter par les instructions entre accolades. Observer que les

sections de code à exécuter sont terminées par l'instruction `break`. Cette dernière permet le passage du contrôle en dehors de `switch-case`. Ainsi, après avoir terminé l'exécution de `instr2`, le programme rencontre l'instruction `break` et il effectue un saut inconditionnel à la ligne immédiatement après la dernière accolade de `switch`. Par contre si `6000 - id = 103`, dans ce cas, le programme exécute `instr3` mais après quoi il poursuit l'exécution de `instr4` puis l'instruction `break`. Donc, le regroupement judicieux des sections de code par les instructions `case` et `break` peut donner des effets très intéressants.

1.10.3 INSTRUCTION FOR

Pour la réalisation des boucles itératives, l'instruction `for` est probablement la plus versatile. Sa syntaxe est la suivante :

```
for (init; expr1; expr2) { instr; }
```

où `init` est une ou plusieurs instructions d'initialisation de variables, `expr1` est une expression d'évaluation, `expr2` est une expression qui modifie les variables initialisées. Tant que `expr1` est évaluée à non nulle, l'instruction (ou instruction composée) `instr` est exécutée. Normalement, `init` sert à initialiser une variable de comptage, `expr1` vérifie si le compteur a atteint sa limite et `expr2` incrémente (ou décrémente) le compteur. Voici un exemple :

```
1. int i, itab[500];
2. for (i = 0; i < 500; i++)
3.   itab[i] = i;
```

La variable `i` sert de compteur. Les lignes 2 et 3 seront exécutées 500 fois consécutivement puisque la condition `i < 500` demeure vraie (non nulle) pour $0 \leq$ `i` ≤ 499 et que le compteur est incrémenté à chaque itération par l'expression `i++`. Ainsi, à la fin de l'exécution, le tableau `itab` aura comme éléments 0 à 499 inclusivement. La déclaration du type du compteur (`int i`) peut être réalisée directement dans la partie initialisation de l'instruction `for`. Cependant, le standard ANSI C++ décourage ce genre de codage. Par souci de portabilité, il est conseillé de déclarer le type des variables en dehors de l'instruction `for`. L'instruction `for` peut accepter des constructions plus complexes. Voici un exemple.

```
int i;
float j, ftab[500][500];
for (i = 0, j = 3.1416; i < 500; ftab[i][i] = i * j, i++) ;
```

Nous avons, à l'aide de l'instruction `for`, rempli la diagonale principale de la matrice `ftab` :

$$\begin{bmatrix} 0 & & \\ & \ddots & \\ & & i \times j \end{bmatrix} \quad \begin{array}{l} i = 0,1,\ldots,N \\ j = 3.1415. \end{array}$$

Tout le travail est réalisé dans la ligne 3. La ponctuation point-virgule (`;`) sert à indiquer qu'il n'y a pas d'instructions supplémentaires à exécuter dans la boucle. Vous pouvez certainement imaginer des constructions plus utiles.

1.10.4 CONSTRUCTION DO-WHILE

Dans la présentation de l'instruction `for`, il est évident que si au démarrage l'expression d'évaluation (`expr1`) est nulle alors le bouclage n'aura pas lieu. Pour s'assurer que le bouclage s'exécute au moins une fois et ce peu importe la condition d'évaluation, on peut employer l'instruction `do-while`. Sa syntaxe est :

```
do { instr } while (expr);
```

où `instr` est une instruction (ou instruction composée) à exécuter et `expr` est une expression. La boucle `do-while` est toujours exécutée au minimum une (1) fois. Le bouclage continue tant que `expr` évalue à non nulle. Par exemple,

```
int itab[500], i = 0;
do {
  itab[i] = i++;
} while (i < 500);
```

est logiquement équivalent à la construction de `for` montrée plus tôt.

1.10.5 INSTRUCTION WHILE

L'instruction `while` peut s'employer seule. Dans ce cas, la condition de bouclage est vérifiée avant la première itération. L'instruction `while` est surtout utile lorsque la limite de bouclage n'est pas connue d'avance. Par exemple,

```
1. int i = 0;
2. while (chaine1[i] != 0) {
3.   chaine2[i] = chaine1[i];
4.   i++;
5. }
6. chaine2[i] = 0;
```

Pourquoi ne peut-on pas simplement copier deux chaînes par une assignation `chaîne1 = chaîne2`?

Les lignes 1 à 6 permettent le copiage des caractères d'une chaîne de caractères dans une autre chaîne. Noter que la ligne 6 est obligatoire afin de terminer correctement une chaîne par un zéro. Tout comme l'instruction `for`, le bouclage n'aura pas lieu si la condition de bouclage est fausse a priori. Évidemment, il existe dans la bibliothèque `string` des fonctions beaucoup plus simples pour effectuer le copiage des chaînes de caractères.

1.11 FONCTIONS

La décomposition d'une routine de traitement en sous-routines est réalisée par des fonctions en C++. D'ailleurs, un programme C++ contient toujours une fonction

principale appelée `main()`[5] dans laquelle d'autres fonctions sont appelées. Une fonction de C++ est réalisée en deux étapes. La première est sa déclaration. Une déclaration de fonction permet au compilateur de connaître l'existence de la fonction et d'effectuer les vérifications de type (*type checking*). La seconde étape consiste à définir le fonctionnement de la fonction. C'est dans la définition que l'on code la logique de la fonction. Une fois ces deux étapes réalisées, nous pouvons l'utiliser en l'invoquant dans notre programme. Voici un exemple :

```
char* puissance(char chaine[], unsigned int n);
```

Observer que les déclarations sont toujours terminées par un point-virgule.

Il s'agit là une déclaration de la fonction `puissance` appliquée à une chaîne de caractères. On indique au compilateur que cette fonction retourne une chaîne de caractères (`char*`) et qu'elle accepte comme paramètres une chaîne de caractères `chaine` et un nombre entier non signé `n`. Les paramètres d'entrée de la fonction sont également appelés la **signature** de la fonction.

```
1. char* puissance(char chaine[], unsigned int n)
2. {
3. unsigned int i, len = strlen(chaine);
4. char* chaine2 = new char[n*len+1];     // allocation dynamique
5. chaine2[0] = 0;                  // créer une chaine nulle
6. for (i=0; i<n; i++)              // concaténer n fois
7.   strcat(chaine2, chaine);
8. return (chaine2);           // retourne la nouvelle chaine
9. }
```

Observer que les définitions de fonction ne sont pas terminées par un point-virgule.

Les lignes 1 à 9 ci-dessus donnent la définition de la fonction `puissance()`. L'opération puissance dans ce contexte est définie comme la concaténation d'une chaîne de caractères. Nous avons créé une nouvelle chaîne *n* fois la longueur obtenue par `strlen()` de la chaîne originale en utilisant l'instruction `new`. La concaténation est réalisée par la fonction `strcat()` de la bibliothèque `string`. La nouvelle chaîne est retournée à l'appelant par l'instruction `return`. Donc, une fonction peut elle-même appeler d'autres fonctions.

```
1. /* traitement */
2. bool ChaineNulle;
3. char* Chaine;
4.  : : :
5.  : : :
6. Chaine = puissance(ch1, 2);
7. if (strlen(Chaine) == strlen(ch1)) {
8.   ChaineNulle = true;
9.   delete [] Chaine;
10. } else
11.   ChaineNulle = false;
12.  : : :
13.  : : :
```

Une fois la fonction déclarée et définie, il est maintenant possible de l'utiliser dans le programme. Par exemple, à la ligne 6, le programme appelle la fonction `puissance()` et emmagasine le résultat retourné dans la variable `Chaine`.

[5] Dans l'environnement Windows, un programme C/C++ possède toujours une fonction principale appelée `Winmain()`.

1.11.1 VISIBILITÉ D'UNE FONCTION

En pratique, il est tout à fait possible qu'une fonction ne soit pas déclarée. La raison est qu'il existe des règles de visibilité gouvernant l'étendue (*scope*) d'une variable ou fonction. En voici quelques règles qu'il faille observer.

Une fonction n'a pas à déclarer son existence au compilateur si sa définition précède son utilisation dans un fichier source.

On doit toujours déclarer l'existence d'une fonction si elle est définie dans un autre fichier source ou si elle provient d'une bibliothèque.

La déclaration d'une fonction doit précéder sa définition et son utilisation dans un fichier source.

L'extension `.h` n'est qu'une convention. En effet, vous pouvez donner n'importe quel nom après la directive `#include`.

C'est pour cette raison que l'on déclare toujours les fonctions au début d'un fichier source avant toute définition. Pour simplifier l'écriture, nous utilisons souvent des fichiers d'en-tête (fichiers `.h`) pour contenir les diverses déclarations. Nous incluons ces fichiers d'en-tête dans le fichier source au lieu de réécrire chacune des déclarations[6]. Reprenons l'exemple au bas de la page précédente et explicitons l'emplacement des déclarations et définitions.

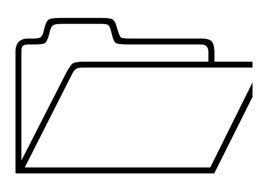
```
/* traitement */
#include <string.h> // déclaration des fonctions de la bibliothèque string
 : : :
// déclaration de la fonction puissance()
char* puissance(char chaine[], unsigned int n);
 : : :
void main() {
bool ChaineNulle;
char* Chaine;
 : : :
 : : :
Chaine = puissance(ch1, 2);
if (strlen(Chaine) == strlen(ch1)) {
  ChaineNulle = true;
  delete [] Chaine;
}
else
  ChaineNulle = false;
 : : :
}


// définition de puissance()
char* puissance(char chaine[], unsigned int n)
{
unsigned int i, len = strlen(chaine);
chaine2 = new char[n*len+1];// allocation dynamique
chaine2[0] = 0;             // créer une chaine nulle
for (i=0; i<n, i++)         // concaténer n fois
 strcat(chaine2, chaine);
return (chaine2);       // retourne la nouvelle chaine
}
```

Pour pouvoir utiliser les fonctions de la bibliothèque string, nous devons inclure le fichier d'en-tête `string.h`. La directive de pré-traitement `#include <string.h>`

[6] Parfois, il peut exister des centaines de déclaration dans une bibliothèque !

indique au compilateur de chercher dans le chemin système, le fichier `string.h`. On aurait pu écrire `#include "string.h"`. Dans ce cas, le compilateur cherchera aussi dans le chemin spécifié par le programmeur. Normalement, on utilise `< >` pour désigner une bibliothèque du système et `" "` pour désigner un fichier d'en-tête créé par le programmeur.



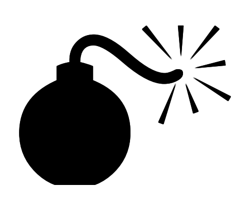
Remarquer que la déclaration de la fonction `puissance()` précède son utilisation dans la fonction `main()`. La définition même de `puissance()` n'est effectuée qu'à la fin du fichier source. En effet, peu importe l'emplacement de la définition, l'importance est que la déclaration doit précéder l'utilisation. On appelle aussi en C++ la déclaration d'une fonction comme le **prototype** de la fonction. Le compilateur utilise le prototype pour effectuer la vérification des paramètres d'une fonction lors de la compilation. Enfin, si la définition de `puissance()` précède la fonction `main()` alors il n'est pas nécessaire de déclarer le prototype. Cependant, nous vous déconseillons fortement cette façon de faire puisqu'elle risque de vous causer des problèmes de visibilité, surtout dans des programmes plus complexes.

1.11.2 FONCTIONS INLINE

Toutes les fonctions de C++ sont considérées comme des sous-routines avec une adresse de retour. Le contrôle de l'exécution est passé momentanément dans la fonction puis revient au point où la fonction a été appelée. Pour réaliser ce va-et-vient, le compilateur doit générer du code supplémentaire connu sous le nom de **prologue** et **épilogue**. Le prologue prépare l'appel de la fonction en empilant les paramètres d'entrée de la fonction, l'adresse de retour dans la pile et charge son adresse dans le compteur d'adresse du processeur. L'épilogue dépile tous les objets de la pile (y compris le résultat de retour et toutes les variables locales utilisées par la fonction) et charge l'adresse de retour dans le compteur d'adresse. Toutes ces opérations exigent du temps de traitement.

On peut indiquer au compilateur qu'il est préférable d'inclure le code exécutable de la fonction directement au point où la fonction est appelée. L'inclusion directe du code exécutable d'une fonction évite la génération de prologue et épilogue. La directive `inline` permet d'accorder ce privilège à une fonction. On le place devant la déclaration d'une fonction.

```
inline char* puissance(char chaine[], unsigned int n);
```



À noter qu'il n'est pas possible d'utiliser une fonction inline lors d'un appel récursif et lors d'un appel utilisant un pointeur de fonction. De plus, une fonction inline peut augmenter considérablement la taille du programme. Veuillez ne pas abuser cette capacité du langage C++.

1.11.3 SURCHARGE DES FONCTIONS

La surcharge (*overload*) des fonctions permet l'utilisation d'un même nom pour identifier différentes fonctions. Le compilateur est en mesure d'identifier la fonction

à utiliser en vérifiant la signature des fonctions. La signature d'une fonction est l'ensemble des paramètres d'entrée de la fonction. Par exemple le code ci-dessous réalise deux fonctions de même nom mais donc la signature est différente.

```
1. int somme(int* vec, unsigned int len)
2. {
3. int i, sum = 0;
4. for (i=0; i < len; i++)
5.   sum += vec[i];
6. return (sum);
7. }
8.
9.
10. int somme(float* vec, unsigned int len)
11. {
12. int i;
13. float sum = 0.0;
14. for (i=0; i < len; i++)
15.   sum += vec[i];
16. return int(sum);
17. }
```

À la ligne 1, la fonction `somme()` accepte un vecteur d'entiers alors que la fonction `somme()` de la ligne 10 accepte un vecteur d'éléments en point flottant. Le type de retour de la fonction ne joue aucun rôle dans la surcharge des fonctions. À l'invocation de `somme()`, le compilateur vérifie la signature de la fonction et les paramètres passés. Il détermine ensuite la bonne fonction à utiliser. Tout ceci est transparent au programmeur. Une fonction peut être surchargée par une autre si les deux fonctions n'ont pas le même type de signature ou si le nombre de paramètres d'entrée est différent.

Le compilateur accepte la surcharge des fonctions parce qu'il effectue toujours le mélange des noms (*name mangling*) dans ses phases internes de compilation. Ce mélange consiste à ajouter des caractères dans les noms de fonction afin de les distinguer les unes des autres. Le mélange des noms est rendu obligatoire à cause de la possibilité d'héritage des classes de C++. Une classe enfant peut très bien posséder une fonction de même nom que celle de son parent. Le compilateur doit pouvoir distinguer ces fonctions sans ambiguïté.

1.11.4 SURCHARGE DES OPÉRATEURS

Nous avons vu que les fonctions peuvent être surchargées. En C++, il est possible de surcharger les opérateurs. La surcharge des opérateurs est simplement la possibilité de donner une autre signification aux opérateurs existant dans le langage. Par exemple, `i++` signifie postincrémenter la variable `i`. Si cette dernière est un entier alors on l'incrémente de 1. On peut très bien créer une classe réalisant une pile et redéfinir l'opérateur `++` pour signifier l'action d'empiler. De même, il est possible de surcharger l'opérateur `--` pour signifier l'action de dépiler.

La surcharge des opérateurs est tellement répandue en C++ que nous l'utilisons sans le savoir. Prenons l'exemple d'entrée-sortie à l'aide des objets `cout` et `cin`. On peut insérer des données dans le flux de sortie à l'aide l'opérateur d'insertion `<<`.

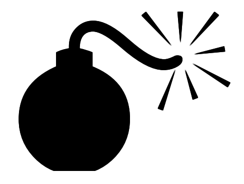
```
cout << 1234 << "Je suis ici!\n";
```

Dans cet exemple l'opérateur `<<` est surchargé dans la classe réalisant `cout` pour réaliser l'insertion des données dans le flux de sortie. Une remarque est nécessaire ici. L'opérateur `<<` est en fait surchargé plus d'une fois dans la classe réalisant `cout`. Les concepteurs ont surchargé `<<` pour chaque type de données acceptées par `cout`. C'est pour cette raison que l'on peut mélanger des entiers (`1234`) et des chaînes (`"Je suis ici!\n"`) en utilisant le même opérateur.

Consulter la section Dérivation de classes à la page 32 pour connaître la syntaxe impliquée dans la surcharge des opérateurs. Vous pouvez surcharger les opérateurs qui sont énumérés dans le Tableau 5.

<table>
<tr><th colspan="3">Opérateurs permettant la surcharge</th></tr>
<tr><td>+</td><td>-=</td><td>++</td></tr>
<tr><td>-</td><td>^=</td><td>--</td></tr>
<tr><td>*</td><td>&=</td><td>(,)</td></tr>
<tr><td>/</td><td>|=</td><td>[,]</td></tr>
<tr><td>%</td><td><<</td><td>new, delete</td></tr>
<tr><td>^</td><td>>></td><td>&</td></tr>
<tr><td>!</td><td><<=</td><td>|</td></tr>
<tr><td>=</td><td><=</td><td>~</td></tr>
<tr><td><</td><td>>=</td><td>*=</td></tr>
<tr><td>></td><td>&&</td><td>/=</td></tr>
<tr><td>+=</td><td>||</td><td>%=</td></tr>
<tr><td>>>=</td><td>==</td><td>!=</td></tr>
<tr><td>,</td><td>-></td><td>->*</td></tr>
</table>

Tableau 5 Les opérateurs qui peuvent être surchargés en C++.



L'opérateur d'assignation (=) est le seul que l'on ne peut pas hériter (voir Dérivation de classes, page 32). Une classe dérivée ne peut utiliser l'opérateur d'assignation de sa classe de base. De plus, l'opérateur d'assignation doit être surchargé par une fonction non statique (voir Membres statiques, page 44). Cette fonction ne peut être une amie de la classe (voir Fonctions amies, page 40).

1.11.5 FONCTION RETOURNANT UNE RÉFÉRENCE

Nous avons vu qu'une variable référence est un alias d'une autre variable. Nous pouvons également utiliser l'opérateur de référence dans le type de retour d'une fonction. Son utilité est plus que surprenante si nous considérons qu'une référence comme type de retour permet à une fonction d'être utilisée comme lvalue (à gauche d'une assignation). Prenons l'exemple suivant.

```
int iVec[10];
 : : :
 : : :
int& NouveauX(int i)
{
return iVec[i];
}

void main()
{
int i;
for (i=0; i<10; i++)
```

```
13.   NouveauX(i) = i;
14. }
```

La fonction `NouveauX()` accepte un entier et retourne une référence à un entier. À chaque invocation de la fonction `NouveauX()`, cette dernière retourne une référence à un des éléments du vecteur `iVec`. Dans la fonction `main()`, on remplit le vecteur `iVec` par une boucle `for`. Observer bien la ligne 14. La fonction `NouveauX()` est utilisée comme lvalue ! Cette ligne assigne la valeur de `i` à l'élément du vecteur `iVec` retourné par `NouveauX()`. Ceci est tout à fait légal puisque `NouveauX()` retourne une référence d'une variable (dans ce cas, un élément de `iVec`).

1.12 CLASSES DE STOCKAGE

Ne pas confondre avec les classes (`class`) de C++ ☞

Chaque variable et fonction possède un type et une classe de stockage. Les classes de stockage sont données par les mots clés: *i*) `auto`; *ii*) `extern`; *iii*) `register`; *iv*) `static`.

1.12.1 STOCKAGE AUTOMATIQUE

Il s'agit de la classe la plus courante. Une variable déclarée dans une instruction composée est une variable automatique. Par exemple pour calculer la somme des éléments d'un vecteur,

```
1. int somme(int* vec, unsigned int len)
2. {
3. int i, sum = 0;
4. for (i=0; i < len; i++)
5.   sum += vec[i];
6. return (sum);
7. }
```

Dans cette fonction, les variables `i` et `sum` sont des variables automatiques. Il n'est pas nécessaire d'ajouter le mot clé `auto` devant la déclaration de `i` et `sum`. Leur contexte étant clair pour le compilateur. Enfin, les variables automatiques sont placées dans la pile de l'environnement d'exécution. C'est pour cette raison qu'une variable automatique n'existe que dans le bloc d'instructions. Ainsi, la ligne 6 retourne le contenu de la variable `sum` et non son adresse.

1.12.2 STOCKAGE EXTERNE

Une variable externe est celle qui est déclarée en dehors des fonctions. Elle est considérée comme variable globale pour les fonctions déclarées par la suite. La durée de vie des variables externes est la même que celle du programme. Dans un fichier source, la déclaration et l'initialisation des variables externes sont semblables à celles des variables automatiques. Il suffit de les placer en dehors des fonctions et préférablement au début du fichier source.

```
/* traitement */
#include <string.h> // déclaration des fonctions string
 : : :
// déclaration de la fonction puissance()
char* puissance(char chaine[], unsigned int n);
 : : :
// variables externes
float FVec[100];
float* pFVec = FVec;  // idem. &FVec[0]

void main() {
// variables automatiques;
int i;
float sum = 0.0;
char* chaine;
 : : :
 : : :
 : : :
 : : :
for (i=0; i<100; i++)
  sum += pFVec[i];   // idem. *(pFVec+i) ou FVec[i]
 : : :
 : : :
chaine = puissance("Je suis ici!", i) ;
 : : :
 : : :
}
```

Quelle est la valeur d'une variable externe qui n'est pas initialisée ?

Dans le code ci-dessous, les variables `FVec` et `pFVec` sont de classe externe. Elles peuvent être utilisées par toutes les fonctions déclarées dans le fichier source.

Fichier source `A.cpp`

```
char* chaine = "Je suis ici!";
: : :
: : :
```

Fichier source `B.cpp`

```
: : :
: : :
extern char* chaine;
: : :
: : :
```

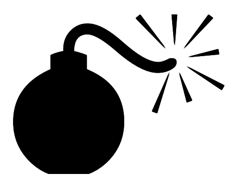
Figure 1 Utilisation du mot clé `extern`.

Si la définition de la variable est réalisée dans un autre fichier source alors il faut précéder à la déclaration d'une variable externe par le mot clé `extern`. La Figure 1 en est un exemple.

La variable externe `chaine` est déclarée et initialisée dans le fichier `A.cpp`. Pour rendre la variable `chaine` visible dans le fichier `B.cpp`, il faut précéder sa déclaration par le mot clé `extern`. Ce mot clé indique au compilateur qu'il faut trouver la définition de `chaine` dans un autre fichier source[7]. La variable `chaine` est

[7] Évidemment, il faut indiquer au compilateur que le fichier `A.cpp` fait partie du projet sans quoi il ne pourra jamais trouver la définition de la variable `chaine`.

visible pour les fonctions du fichier `B.cpp` qui sont déclarées après `extern char* chaine`.



Éviter l'utilisation des variables externes car elles violent le principe d'encapsulation de la programmation orientée objet. De plus, les effets secondaires sont difficiles à contrôler. Une variable qui est globale est, par définition, accessible à tous. Sa valeur peut être modifiée par toutes les fonctions du fichier source et même les fonctions des autres fichiers sources.

1.12.3 STOCKAGE REGISTRE

Prenons l'exemple suivant :

```
1. void main() {
2.   int i = 0;
3.   while (i < 100) {
4.     chaine = puissance(ch1, i);
5.     i++;
6.   }
7. }
```

La variable `i` sert de compteur et est initialisée à la ligne 2. Elle est utilisée dans la comparaison à la ligne 3 et est incérmentée à la ligne 5. La variable `i` est de classe automatique puisque déclarée à l'intérieur d'un bloc d'instructions. Elle dispose d'une adresse mémoire. Cela signifie que chaque fois `i` intervient dans le code, le compilateur doit générer les instructions, en langage assemblé, pour la charger de la mémoire dans un des registres du processeur. Le processeur effectue les opérations sur le contenu de la variable puis le dépose à son adresse mémoire. Prenons la ligne 5 où nous effectuons la postincrémentation de `i`. Le processeur charge le contenu de `i` dans un registre, effectue son incrémentation puis dépose la valeur obtenue dans la case mémoire de `i`. Ces opérations de transfert exigent du temps de processeur.

Le mot clé `register` indique au compilateur de placer, si possible, la variable désignée dans un registre du processeur. Éliminant ainsi le transfert mémoire-registre de la variable. Le mot clé est une indication de notre souhait. Le compilateur peut ne pas respecter ce souhait. Ainsi, le code révisé peut ressembler à ceci :

```
1. void main() {
2.   register int i = 0;
3.   while (i < 100) {
4.     chaine = puissance(ch1, i);
5.     i++;
6.   }
7. }
```

La ligne 2 indique au compilateur que l'on souhaite placer au préalable `i` dans un des registres du processeur. Si le compilateur est en mesure de le faire, il placera `i` dans un registre libre et générera les instructions nécessaires pour accéder `i` directement à partir du registre utilisé. L'utilisation du mot clé `register` peut accélérer l'exécution des routines.

1.12.4 STOCKAGE STATIQUE

Cette classe sert à modifier la durée de vie et à restreindre la visibilité des variables. Lorsque le mot clé `static` est appliqué à une variable automatique, la durée de vie de la variable devient la même que celle du programme. Rappelons qu'une variable automatique n'existe que dans un bloc d'instructions. Elle est placée dans une pile de l'environnement d'exécution. Le mot clé `static` permet la modification de la durée de vie d'une variable en forçant le compilateur à la placer dans la région de données (`data`) de l'environnement d'exécution. Une variable automatique statique conserve son contenu même si l'exécution du bloc d'instructions est terminée.

```
1. int somme(int* vec, unsigned int len)
2. {
3. int i;
4. static int sum = 0;
5. for (i=0; i < len; i++)
6.   sum += vec[i];
7. return (sum);
8. }
```

Dans cette routine de sommation, la variable automatique `sum` est modifiée en classe statique. Elle va pouvoir conserver sa valeur entre les appels de la fonction `somme()`. On peut donc réaliser une somme cumulée par cette utilisation de la variable `sum`.

L'initialisation des variables de classe statique est toujours effectuée une seule fois, au moment de sa première utilisation. Ainsi, la variable `sum` n'est initialisée à zéro qu'au premier appel de la fonction `somme()`. La ligne 4 n'est plus exécutée pour les appels subséquents de `somme()`.

Quand une variable est de classe externe, le modificateur `static` enlève sa visibilité aux autres modules du programme. En d'autres mots, une variable globale et statique n'est utilisable que dans le fichier source où elle a été déclarée. On dit qu'elle est privée au module[8].

1.13 STRUCTURE ET UNION

Une structure et union permettent l'assemblage de type hétéroclite en une seule entité. Elles sont normalement utilisées pour contenir des données du programme. Au lieu d'avoir des variables éparpillées partout dans le code, il est souvent plus intéressant de les regrouper dans des structures ou union. La déclaration d'une structure est comme suite :

```
struct adresse {
  char Nom[80];   // nom de l'occupant
int NumeroCivique;
  char Rue[120];
  char Ville[80];
  char Province[4];
  char CodePostal[7];
}
```

[8] Ne pas confondre avec le niveau d'accès privé des classes.

On voit qu'une structure est composée d'un nombre de champ. Le type des champs peut être les types de base, les structures, les unions et les objets en C++. On peut créer une instance d'une structure définie et l'assigner à des données par :

```
1. adresse MonAdresse;
2. strcpy(MonAdresse.Nom, "Jean Etudiant");
3. strcpy(MonAdresse.Rue, "rue Notre-Dame Ouest");
4. strcpy(MonAdresse.CodePostal, "H4L 1K8");
5. strcpy(MonAdresse.Province, "QUE");
6. strcpy(MonAdresse.Ville, "Montréal");
7. MonAdresse.NumeroCivique = 1234;
```

La ligne 1 crée une instance de structure `adresse`. Les lignes 2 à 7 effectuent l'assignation des champs. Remarquer que cette assignation n'a pas besoin d'être ordonnée. On peut aussi initialiser une instance d'une structure en même temps que sa création.

```
adresse MonAdresse = {"Jean Etudiant", 1234, "rue Notre- Dame Ouest",
   ↳"Montréal", "QUE", "H4L 1K8"};
```

Dans ce cas, les assignations doivent respecter l'ordre des champs de la structure. Il est également possible d'inclure des fonctions parmi les champs d'une structure. On peut ainsi obtenir une classe C++ dont le niveau d'accès est public pour les données (champs) et les fonctions de la structure.

L'union est déclarée de la même façon qu'une structure. La différence est dans l'interprétation des champs par le programmeur. Prenons l'exemple suivant :

```
union MonType {
  long ltype;       // un entier long
  float ftype;      // un simple précision
  double dtype;     // un double précision
};
```

Il existe trois champs dans cette union. Par contre, le compilateur ne réserve qu'un espace suffisant pour contenir le type le plus grand. Dans notre cas, c'est la variable à double précision (8 octets). Avec les unions, on ne peut qu'utiliser un champ à la fois. Pour notre exemple, nous pouvons placer un entier long, un nombre simple précision ou un nombre double précision, mais pas les trois en même temps. L'intérêt des unions est dans le fait que, dans certaines circonstances, le programme a besoin d'un lieu pour entreposer momentanément une donnée. On peut donc conserver l'espace mémoire en créant une union pour entreposer ces données momentanées. L'usage des unions n'est plus aussi important puisque aujourd'hui, la quantité de mémoire des ordinateurs dépasse les millions octets.

Pour la programmation de bas niveau, l'union est souvent utilisée pour extraire la représentation binaire des nombres. Par exemple, on peut placer une valeur de double précision dans le champ `dtype` et retirer sa représentation binaire par le champ `ltype`.

```
MonType montype;
montype.dtype = 123456.12;   // placer un double
int binaire = montype.ltype; // retirer un long int
```

1.14 POINTEURS

Cette sous-section est à lire attentivement ☞

Les pointeurs sont indispensables dans un programme C++. On l'associe souvent à la mémoire allouée dynamiquement. Or, il suffit que l'on dispose d'une adresse mémoire et l'application des pointeurs devient possible.

Un pointeur est une variable pouvant contenir une adresse mémoire. Pour pouvoir manipuler correctement un pointeur, on doit aussi spécifier le type de donnée pointé. Par exemple, pour contenir une adresse d'une variable entière, nous devons indiquer le type de la variable devant le pointeur. Par exemple,

```
1. int *pI;              // pointeur de int
2. float *pF;            // pointeur de float
3. double *pD;           // pointeur de double
```

On ne peut que mettre des adresses mémoires dans un pointeur. Pour obtenir l'adresse d'une variable, nous devons utiliser l'opérateur `&` (adresse-de). Donc, pour assigner l'adresse d'une variable à un pointeur, il faut respecter le type. Ainsi,

```
1. int X;
2. float Y;
3. double Z;
4.
5. int *pI = &X;
6. float *pF = &Y;
7. double *pD = &Z;
8. int *pI = &Y;         // Erreur ! type incompatible
```

Pour obtenir le contenu d'une variable pointée par le pointeur, nous devons utiliser l'opérateur `*` (indirection). Ainsi, selon l'exemple ci-dessus :

```
9. *pI = 12;                 // X = 12
10. *pF = 3.1416;             // Y = 3.1416
11. *pI++;                    // X = 13
12. *pF = 2.0 * (*pF);        // Y = 6.2832
```

Remarquer que l'indirection permet à une variable d'être un lvalue (à gauche d'une assignation) et un rvalue (à droite d'une assignation). La ligne 12 présente un fait intéressant. L'opérateur multiplication et l'opérateur d'indirection partagent le même symbole (`*`). Le compilateur reconnaît toujours l'indirection comme l'opération la plus prioritaire, il va donc substituer d'abord `*pF` par la valeur de `Y` avant d'effectuer la multiplication avec la constante `2.0`. Mais pour alléger la lecture du code source, il est recommandé de regrouper l'indirection entre parenthèses. On peut aussi créer un pointeur de type `void`. Le type `void` représente un type indéfini. Un pointeur de type indéfini peut accepter tout autres les types. Par exemple,

```
13. void *pV;
14. pV = &X;                  // ok
15. pV = &Y;                  // ok
16. pV = &Z;                  // ok
```

Or, pour pouvoir utiliser correctement le contenu pointé, vous devez effectuer une conversion (*cast*) sur le pointeur d'abord.

```
17.*((double*)pV) = 1.0;          // Z = 1.0
```

La signification de la ligne 17 est la suivante : le pointeur `pV` est converti en pointeur de double précision `((double*)pV)`. La constante `1.0` est assignée à la variable pointée `(*((double*)pv))`. La conversion d'un type de pointeur `void` est nécessaire pour le compilateur. Prenons l'exemple suivant :

```
18.int XX = *pI;
19.float YY = *pF;
20.double ZZ = *pD / 5.1234;
```

À la ligne 18, le programme doit copier 2 octets (pointé par `pI`) de données dans `XX`[9]. Le programme copie 4 octets (pointé par `pF`) dans `YY`. À la ligne 20, le programme transfert 8 octets de la mémoire (pointé par `pD`) à un des registres du processeur. Il effectue un calcul en double précision et transfert le résultat (8 octets) dans la mémoire (pointé par `pD`). Le type du pointeur est donc important pour le bon fonctionnement du programme. Il explique également pourquoi nous devons convertir un pointeur `void` avant son utilisation.

1.15 Pointeurs et tableaux

Les pointeurs et les tableaux sont des objets très reliés. En effet, le nom d'un tableau est le pointeur même du tableau.

```
#include <iostream.h>

const float PI = 3.141516;
void main()
{
int i;
float fVec[4] = {PI, PI*2, PI*3, PI*4};

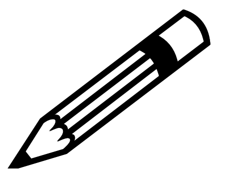
for (i=0; i<4; i++) {
   cout << fVec[i] << endl;     // version utilisant [ ]
   cout << *(fVec+i) << endl;  // version avec pointeur
}
```

L'exécution de ce programme donne le résultat :

```
3.141516

3.141516
6.283032
6.283032
9.424548
9.424548
12.566064
12.566064
```

[9] La grandeur d'un entier est de 2 octets pour un ordinateur PC.



Chaque élément de `fVec` est affiché deux fois tour à tour. Le premier affichage est réalisé par `fVec[i]`, le deuxième par `*(fVec+i)`. Donc, `fVec` est le nom du tableau et est aussi un pointeur et son type est `float`. La différence entre un pointeur *ordinaire* et un nom de tableau est que ce dernier joue le rôle d'un pointeur constant. En effet, on ne peut pas obtenir une construction du genre `*(fVec++)`. `fVec` est une constante et on ne peut pas l'incrémenter. L'exemple ci-dessous explique cette différence.

```
1. #include <iostream.h>
2.
3. const float PI = 3.1415;
4. void main()
5. {
6. int i;
7. float fVec[4] = {PI, PI*2, PI*3, PI*4};
8. float *pfVec = fVec;
9.
10. for (i=0; i<4; i++) {
11.   cout << fVec[i] << endl;     // version utilisant [ ]
12.   cout << *(fVec++) << endl;  // Erreur ! ! ! !
13.   cout << *(pfVec++) << endl; // légal !
14. }
```

Si la ligne 12 est laissée dans le code source, le programme ne pourra être compilé. La ligne 12 est une expression erronée. Par contre, la ligne 13 est tout à fait légale puisque `pfVec` est une variable et non une constante comme l'est `fVec`. À noter que l'on pourrait aussi écrire la ligne 8 comme suite :

```
8. float *pfVec = &fVec[0];
```

La raison est que `fVec` est équivalent à `&fVec[0]`. D'abord, `fVec[0]` donne le contenu de la case de mémoire du premier élément du tableau. Par application de l'opérateur `&`, on obtient l'adresse de cette case de mémoire. La Figure 2 est une représentation de ces différentes notions.

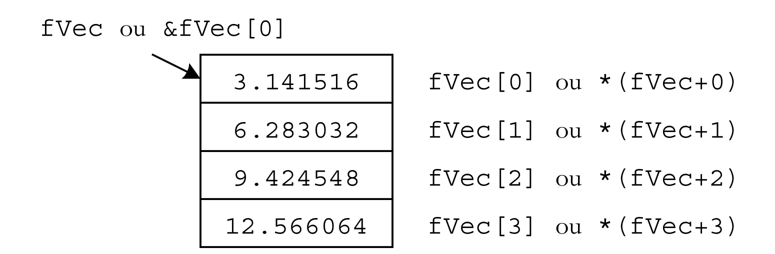


Figure 2 Tableau et pointeur.

1.16 POINTEURS DE FONCTIONS

Il est possible d'assigner, à un pointeur, l'adresse d'une fonction. L'idée est que l'on peut exécuter n'importe quelle fonction de même signature et de même type de retour à l'aide d'un pointeur de fonction. Voici un exemple,

```
1. char foo(int a, float b, double c);   // fnct quelconque
```

```
2. int foobar(char str[], char (*pFnct)(int, float, double));
```

La première déclaration est le prototype d'une fonction appelée `foo`. La deuxième déclaration est le prototype d'une fonction acceptant comme paramètres d'entrée, une chaîne de caractères et un pointeur de fonction. Étudions de plus près ce pointeur de fonction.

```
char (*pFnct)(int, float, double)
```

On peut identifier facilement le type de retour de ce pointeur de fonction, il s'agit d'un `char`. Les arguments de la fonction pointée sont `(int, float, double)`. Le nom du pointeur est `pFnct`. Donc, ce pointeur accepte l'adresse d'une fonction qui retourne un `char` et demande un `int`, un `float` et un `double` comme paramètres d'entrée. La déclaration d'une telle fonction est donnée dans la ligne 1 de l'exemple.

Parfois, l'explication de certaines constructions C++ peut sembler très compliquée. Cependant, la syntaxe impliquée est souvent très simple. Le contraire est aussi vrai.

L'initialisation d'un pointeur de fonction est très simple. Mais d'abord, soulignons le point suivant. Pour un tableau, son nom est évalué à l'adresse de la première case de mémoire (voir la section 1.15 à la page 23). Pour une fonction telle

```
char foo(int a, float b, double c);
```

son nom est évalué à

```
char (*) (int, float, double);
```

Ce qui veut dire que `foo()` est une fonction qui peut être assignée à un pointeur de fonction de même signature et de même type de retour. Voyons maintenant l'initialisation du pointeur de fonction `pFnct`.

```
char (*pFnct)(int, float, double) = foo;
```

L'initialisation du pointeur de fonction `pFnct` est simple et élégante. Il suffit que l'on s'assure que la fonction à assigner au pointeur possède la même signature et le même type de retour.

L'invocation d'une fonction à partir de son pointeur est également très simple. L'opérateur d'indirection (`*`) n'est pas nécessaire dans ce cas. Ainsi,

```
pFnct(6, 1.2, 3.141516e300);
```

est une invocation de la fonction `foo()` par son pointeur `pFnct`. Puisque la syntaxe d'invocation est la même, pourquoi utilise-t-on un pointeur de fonction ? La réponse à cette question est donnée dans l'exemple ci-dessous. Supposons que nous voulions appliquer un algorithme de tri pour ordonner un ensemble de données. Nous savons que certains algorithmes donnent un meilleur rendement que d'autres. Cependant, le rendement des algorithmes de triage est une fonction de la taille des données. Pour donner une plus grande souplesse au programme, nous avons réalisé quatre (4)

algorithmes de triage. Dépendant de la taille des données, l'un de ces algorithmes choisi est exécuté.

```
1.  : : :
2.  : : :
3.// algorithmes de tri. Réalisés dans un autre fichier
4.

5.extern void QSort(int* iVec, int low, int high);
6.extern void MSort(int* iVec, int low, int high);
7.extern void HSort(int* iVec, int low, int high);
8.extern void BSort(int* iVec, int low, int high);
9.  : : :
10.// un tableau de pointeurs de fonction
11.void (*SFunc[])(int*, int, int) = { QSort, MSort, HSort, BSort };
12.  : : :
13.  : : :
14.void main()
15.{
16.int indice;
17.  : : :
18.  : : :
19.// exécuter un algorithme en fonction de la taille des données
20.
21.if (Taille <= 1000)
22.   indice = 0;       // Qsort
23.else if (Taille <= 10000)
24.   indice = 1;       // Msort
25.else if (Taille <= 500000)
26.   indice = 2;       // Hsort
27.else
28.   indice = 3;       // Bsort
29.
30.SFunc[indice](Vec, lo, hi);
31.  : : :
32.  : : :
33.}
```

Les quatre (4) algorithmes de tri sont déclarés `extern`. Cela veut dire qu'ils sont réalisés dans un autre module. Enlever le mot clé `extern` s'ils sont définis dans le même fichier source. La ligne 11 est la déclaration et l'initialisation d'un tableau de pointeurs de fonction. Ces pointeurs ont la même signature et le même type de retour que les fonctions `QSort()`, `Msort()`, `HSort()` et `BSort()`. Dans la fonction principale `main()`, nous déterminons la taille des données à trier. On utilise une variable `indice` pour indiquer la fonction de tri à utiliser. Cette variable sert d'indice dans le tableau des pointeurs de fonction. Enfin, à la ligne 30, on exécute la fonction de tri indiqué par `indice`.

Donc, au lieu d'exécuter la fonction de tri dans l'instruction `if-else`, on a regroupé les points d'invocation des fonctions en une seule ligne. Cette façon de faire réduit la taille du programme et simplifie la gestion. Surtout lorsqu'il y a un grand nombre de fonctions à traiter. Imaginer un programme qui doit reconnaître un fichier de commandes dans lequel des centaines de commandes sont possibles. On peut facilement regrouper les fonctions réalisant les commandes dans des tableaux semblables à celui de l'exemple. Un mécanisme de détection et de recherche peut alors être utilisé pour exécuter la fonction correspondant à la commande lue du fichier.

1.17 ALLOCATION DYNAMIQUE

L'environnement d'exécution C++ (*runtime environment*) met à la disposition des programmes une zone de mémoire libre (*free store*) pour l'entreposage des données. La réclamation de cette zone libre est réalisée par l'allocation dynamique de la mémoire. C'est-à-dire, la réservation de la mémoire lorsque le programme est en exécution. L'allocation dynamique de la mémoire implique nécessairement l'utilisation des pointeurs. Le mot clé `new` sert à allouer une partie de cette zone libre tandis que le mot clé `delete` permet le retour de la mémoire allouée au système. Dans l'exemple de la sous-section précédente, les algorithmes exigent comme paramètres d'entrée :

```
void QSort(int* iVec, int low, int high);
```

un vecteur et deux scalaires (des entiers). Le vecteur peut être alloué statiquement par :

```
int vecEntier[10];
```

ou alloué dynamiquement par :

```
int *vecEntier = new int[10];
```

Le point à remarquer est que l'allocation statique est réalisée lors de la compilation du programme. Alors que l'allocation dynamique est effectuée lors de l'exécution du programme. Ainsi, l'allocation statique demande une expression constante intégrale (constante entière) entre les crochets. Cette restriction n'est pas appliquée dans le cas d'une allocation dynamique[10]. Dans l'exemple ci-dessous, la ligne 7 produira une erreur de compilation. On ne peut pas utiliser une expression non constante (`i*2`) entre les crochets d'une allocation statique. Par contre, la ligne 8 est parfaitement légale puisqu'il s'agit d'une allocation dynamique.

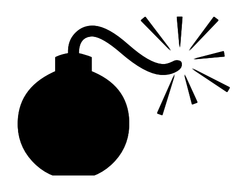
```
1. const int DIM = 10;
2.
3. void main()
4. {
5. int i = 5;
6. int  iVec[DIM];             // ok
7. int *iVec2[i*2];            // Non ! Expr. constante seulement
8. int *iVec3 = new int[i*2];  // ok
9.

10. for (i=0; i<DIM; i++) {
11.   iVec[i] = i*DIM;
12.   iVec3[i] = i;
13. }
14. delete [] iVec3;
15. }
```

[10] Évidemment, l'expression entre crochets doit évaluée à un type intégral (entier).

La syntaxe pour le retour de la mémoire à la zone libre est différente. Si `x` est un scalaire : `delete x`. Si `x` est un tableau `delete [] x`.

À la fin de son utilisation, la mémoire occupée par le vecteur `iVec3` est redonnée à la zone libre par la directive `delete [] iVec3`. Pour un tableau, il est nécessaire de mettre les crochets après le mot clé `delete`. Il ne faut pas mettre les accolades pour des variables scalaires. Les crochets sont une aide au compilateur pour mieux distinguer le contexte.

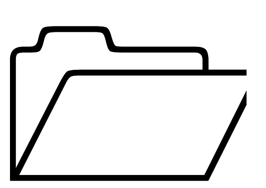


Il existe également une autre façon d'effectuer l'allocation dynamique de la mémoire : `malloc` et `free`. Elle provient du langage C. Éviter son utilisation puisqu'il demande une conversion explicite (*casting*) et sa gestion interne est différente de celle utilisée par `new`. De plus, il est strictement interdit et déconseillé d'employer `free` pour redonner la mémoire au système qui avait été allouée par `new`. Le mélange de ces deux méthodes d'allocation peut produire des erreurs catastrophiques.

1.18 CLASSES ET OBJETS

Une classe est déclarée de la manière suivante :

```
class MaClasse
{
public:
 : : : // variables et fonctions
 : : :
protected:
 : : : // variables et fonctions
 : : :
private:
 : : : // variables et fonctions
 : : :
};
```



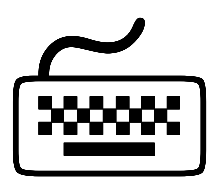
Tout comme une structure, on peut créer une instance (un objet) d'une classe par une déclaration de type : `MaClasse uneclasse`. Le nom `MaClasse` est le type et la variable `uneclasse` est un objet de `MaClasse`. Les variables et fonctions à l'intérieur d'une classe sont appelées **variables membres** et **fonctions membres**. Une classe possède trois niveaux d'accès : *i*) public; *ii*) protégé; *iii*) privé. Une variable ou fonction membre déclarée dans une section identifiée `public` est accessible à tous. Une variable ou fonction membre déclarée dans une section identifiée `protected` est accessible à tous les membres de la classe et ses descendantes. Une variable ou fonction membre déclarée dans une section identifiée `private` n'est accessible qu'aux membres de la même classe.

En pratique, on met uniquement les fonctions d'interface dans la section publique. Habituellement, aucune donnée n'est exposée dans la section publique. Les fonctions membres publiques sont responsables de placer et retourner les valeurs des variables de l'objet. On dit que l'objet possède une interface de services vers le monde extérieur.

La section protégée sert surtout à donner les fonctionnalités nécessaires aux descendants de la classe. Par le mécanisme d'héritage un enfant peut utiliser les

données et les fonctions membres protégées des parents[11]. Enfin, la section privée est surtout utilisée pour les données et fonctions membres qui sont nécessaires pour le bon fonctionnement de la classe. Elles ne sont pas disponibles aux descendantes de la classe.

Dans un programme C++, les classes sont traitées comme des types et les objets sont traités comme des variables. Il est donc possible de les utiliser dans des fonctions ordinaires. Voyons comment nous pouvons utiliser un objet dans un programme C++.



```
#include <iostream.h> // bibliothèque de flux E/S

// définition de la classe Compteur
class Compteur
{
private:
  int compte;
  int mincompte;
  int maxcompte;

public:
  Compteur(int cpt, int min, int max) // constructeur
  Compteur()                          // constructeur
  { compte = 0; mincompte = 0; maxcompte = 100; }
  ~Compteur() { }                     // destructeur
  void AugmenteCompte();
  void DiminueCompte();
  int PrendreComptage();
} ;


// Programme principal
void main()
{
Compteur cpt1, cpt2(0, -10, 20);

cout << "Compteur 1 =" << cpt1.PrendreComptage();
cout << "Compteur 2 =" << cpt2.PrendreCompage();

cpt1.AugmenteCompte(); cpt2.AugmenteCompte();
cpt1.DiminueCompte(); cpt2.AugmenteCompte();

cout << "Compteur 1 =" << cpt1.PrendreComptage();
cout << "Compteur 2 =" << cpt2.PrendreCompage();
}

// constructeur
Compteur::Compteur(int cpt, int min, int max)
{
compte = cpt;
mincompte = min;
maxcompte = max
}

void Compteur::AugmenteCompte()
{
compte++;
compte = (compte > maxcompte) ? maxcompte : compte;
}

void Compteur::DiminueCompte()
{
compte--;
compte = (compte < maxcompte) ? mincompte : compte;
```

[11] Par définition un enfant peut également utiliser fonctions membres publiques des parents.

```
55.}
56.
57.int Compteur::PrendreComptage()
58.{
59.return (compte);
60.}
```

Constructeur et destructeur ☞

Dans ce petit programme, nous avons créé une classe appelée `Compteur`. La déclaration de cette classe est donnée au début du code source. Il existe cinq fonctions membres dans cette classe. Deux fonctions membres spéciales sont désignées comme **constructeur** et **destructeur** de l'objet. Le constructeur est invoqué pour créer une instance d'une classe. Le destructeur est appelé par le compilateur pour effectuer la terminaison de l'instance. Ils n'ont pas de type de retour.

L'unique donnée de la classe `Compteur` est une variable entière `compte`. Elle est placée dans la section privée de la classe. Ainsi, aucun accès possible en dehors de la classe `Compteur`. Pour pouvoir manipuler la variable `compte`, il faut utiliser les fonctions membres publiques. Par la lecture de l'interface publique, on peut constater qu'il est possible d'incrémenter et décrémenter le compteur. Il est aussi possible de lire la valeur du compteur.

Les lignes 30 à 60 correspondent à la définition des fonctions membres de la classe `Compteur`. La définition des fonctions membres possède une syntaxe spéciale. En effet il est toujours nécessaire d'inclure le nom de la classe dans la définition des fonctions membres. La raison est qu'il peut exister, dans le même fichier source, des fonctions de même nom appartenant à des classes différentes. Le constructeur et le destructeur n'ont pas de type de retour. On peut aussi définir les fonctions membres directement dans la déclaration de la classe. Les fonctions membres ainsi créées sont équivalentes à des fonctions inline (voir la sous-section 1.11.2 à la page 14).

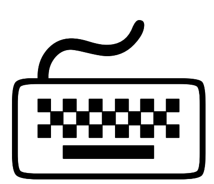
Observer l'utilisation des classes dans le programme principal (ligne 23 à 35). Le programme débute par la création de deux objets de type `Compteur`. Le premier objet utilise la construction sans paramètres, le second utilise le constructeur avec paramètres explicites. Une classe peut disposer autant de constructeurs qu'il est nécessaire. En pratique, on doit prévoir les constructeurs en fonction de la nature d'utilisation de la classe. Par exemple, dans certaines circonstances, il n'est pas facile de déterminer d'avance tous les paramètres d'un objet. Il est alors plus simple d'avoir un constructeur sans paramètres pour la création de l'objet. Plus tard, nous pouvons utiliser l'interface publique pour remplir les paramètres manquants.

Le fonctionnement de ce programme est très simple. Il suffit de mentionner que l'objet `cout` est utilisé pour diriger les données vers la sortie standard qui est l'écran[12]. L'opérateur `<<` représente l'insertion des données dans le flux de sortie. Pour pouvoir utiliser ces objets et les opérateurs associés, il faut inclure le fichier d'en-tête `iostream.h`.

---

[12] On peut aussi rediriger la sortie standard vers un fichier.

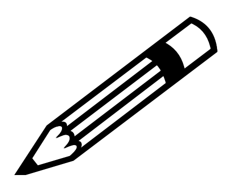
Une question peut se poser. Pourquoi le destructeur de la classe `Compteur` n'est-il pas utilisé explicitement dans le code ? La réponse à cette question est fort simple. Le destructeur (la fonction `~Compteur()`) est appelé par l'environnement d'exécution lorsque la vie de l'objet prend fin. Dans notre exemple, le destructeur des objets `cpt1` et `cpt2` est automatiquement exécuté après la ligne 35 (à la fin de la fonction `main()`). Le destructeur sert à enclencher le processus de terminaison d'un objet. Nous allons modifier notre exemple pour rendre le destructeur plus utile.



```
1.#include <iostream.h> // bibliothèque de flux E/S
2.
3.// définition de la classe Compteur
4.class Compteur
5.{
6.private:
7.  int* compte;
8.  int mincompte;
9.  int maxcompte;
10.
11.public:
12.  Compteur(int cpt, int min, int max) // constructeur
13.  Compteur()                          // constructeur
14.  { compte = 0; mincompte = 0; maxcompte = 100;
15.    compte = new int} // allocation dynamique
16.  ~Compteur()                         // destructeur
17.  void AugmenteCompte();
18.  void DiminueCompte();
19.  int PrendreComptage();
20.} ;
21.
22.// Programme principal
23.void main()
24.{
25.Compteur *cpt1 = new Compteur;
26.Compteur *cpt2 = new Compteur(0, -10, 20);
27.
28.cout << "Compteur 1 =" << cpt1->PrendreComptage();
29.cout << "Compteur 2 =" << cpt2->PrendreCompage();
30.
31.cpt1->AugmenteCompte(); cpt2->AugmenteCompte();
32.cpt1->DiminueCompte(); cpt2->AugmenteCompte();
33.
34.cout << "Compteur 1 =" << cpt1->PrendreComptage();
35.cout << "Compteur 2 =" << cpt2->PrendreCompage();
36.
37.// destruction explicite des objets
38.delete cpt1, cpt2;
39.}
40.
41.// constructeur
42.Compteur::Compteur(int cpt, int min, int max)
43.{
44.compte = new int;
45.*compte = cpt;
46.mincompte = min;
47.maxcompte = max
48.}
49.
50.// destructeur
51.Compteur::~Compteur()
52.{
53.delete compte;
54.}
55.
56.void Compteur::AugmenteCompte()
57.{
58.*compte++;
59.*compte = (*compte > maxcompte) ? maxcompte : *compte;
```

```
60.}
61.
62.void Compteur::DiminueCompte()
63.{
64.*compte--;
65.*compte = (*compte < maxcompte) ? mincompte : *compte;
66.}
67.
68.int Compteur::PrendreComptage()
69.{
70.return (*compte);
71.}
```

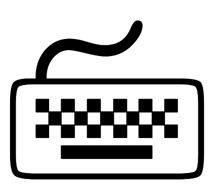
La variable `compte` est maintenant un pointeur d'entier. Nous avons modifié les constructeurs pour allouer dynamiquement l'espace mémoire pour cette variable. Nous devons ajouter les modifications nécessaires pour la manipulation du pointeur `compte` dans les fonctions membres.



Les objets `cpt1` et `cpt2` sont maintenant créés dynamiquement par l'instruction `new`. Observer bien la syntaxe utilisée. Puisque les objets sont maintenant représentés par des pointeurs, leur utilisation demande l'opérateur `->` (le nom de l'opérateur `->` est l'opérateur **sélection-membre**). Nous avons également ajouté une instruction pour la destruction des objets `cpt1` et `cpt2` à la ligne 38. En C++, vous devez toujours détruire les objets créés par l'instruction `new`. À l'exécution de l'instruction `delete` (ligne 38), le destructeur des objets est automatiquement invoqué. Dans cet exemple, le destructeur enlève l'espace mémoire pointé par le pointeur `compte`.

1.19 DÉRIVATION DE CLASSES

Pour créer un lien de parenté entre les classes, nous devons utiliser une déclaration spéciale. Pour démontrer l'**héritage** dans le langage C++, nous allons apporter quelques modifications à la classe `Compteur`.



```
/* Fichier Compteur.h */
#include <iostream.h> // bibliothèque de flux E/S

// définition de la classe Compteur
class Compteur
{
protected:
int compte;

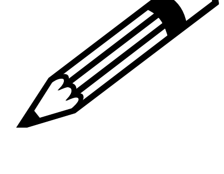
public:
Compteur()
  { compte = 0;}                      // constructeur
  Compteur(int c)                     // constructeur
  { compte = c ; }


  void Raz() { compte = 0; }       // Remise à zéro
  int PrendreComptage()
  { return (compte); }
  Compteur operator ++()              // préincrément
  { return Compteur(++compte); }
  Compteur operator ++(int)
  { return Compteur(compte++); }      // postincrément
};
```

Dans cette déclaration de `Compteur`, nous avons simplifié sa composition tout en réalisant la surcharge de l'opérateur `++` (ligne 18-20). La variable membre `compte` est déplacée de la section privée à la section protégée. La déclaration d'une surcharge d'opérateur est de forme :

```
type operator symbole (param) { instr }
```

où `type` est un type prédéfini du C++ ou une classe quelconque. Le mot clé `operator` est obligatoire. Le symbole est celui qui sera surchargé et `param` représente l'unique paramètre utilisé dans le bloc d'instruction `instr`. **Dans la surcharge des opérateurs, on ne peut avoir plus de deux paramètres entre parenthèses**. Voici les règles à suivre :



- Si un opérateur est à la fois unaire et binaire (par exemple, l'opérateur +), on peut le surcharger deux fois : Une fois pour son utilisation unaire et une fois pour son utilisation binaire.
- Si un opérateur unaire est surchargé par une fonction membre d'une classe, la déclaration de l'opérateur ne prend aucun paramètre. Par contre, la déclaration prend un seul paramètre d'entrée si la surcharge est réalisée par une fonction amie de la classe.
- Si un opérateur binaire est surchargé par une fonction membre d'une classe, la déclaration de l'opérateur prend un seul paramètre d'entrée. Par contre, la déclaration prend deux paramètres d'entrée si la surcharge est réalisée par une fonction amie de la classe.
- Pour les opérateurs qui peuvent être à la fois préfixes et suffixes (++,--), la déclaration de l'opérateur prend aucun paramètre d'entrée pour indiquer son utilisation en préfixe. Par contre, la déclaration prend un seul paramètre d'entrée si la surcharge est réalisée pour une utilisation en suffixes.

Dans le cas du compteur, la surcharge de l'opérateur permet le code suivant :

```
Compteur cpt;            // créer un objet de type Compteur
cpt++;                   // postincrémenter le compteur
```

Ainsi, au lieu de faire appel à une fonction membre pour incrémenter le compteur, on peut utiliser l'opérateur `++`. De cette façon, le programme est plus simple à comprendre.

Le compteur réalisé ne fait qu'incrémenter. On peut créer un nouveau compteur capable d'incrémentation et de décrémentation simplement en dérivant à partir de la classe `Compteur`.

Enregistrer cette déclaration dans le fichier : `Compteur2.h`

```
/* Fichier Compteur2.h */
#include <iostream.h> // bibliothèque de flux E/S
#include "Compteur.h" // en-tête de classe compteur

// définition de la classe Compteur
```

```
6. class Compteur2 : public Compteur
7. {
8. public:
9.   Compteur2() : Compteur()            // constructeur
10.   { }
11.   Compteur2(int c) : Compteur(c)      // constructeur
12.   { }
13.   void Raz() { compte = 0; }       // Remise à zéro
14.   Compteur2 operator --()             // prédécrément
15.   { return Compteur2(--compte); }
16.   Compteur2 operator --(int)
17.   { return Compteur2(compte--); }     // postdécrément
18.
19. };
```

La classe `Compteur2` est dérivée de la classe `Compteur`. La ligne 6 est la déclaration nécessaire pour cette dérivation. Le mot clé `public` indique qu'il s'agit d'un héritage public. La classe `Compteur2` hérite donc du contenu de la section publique et protégée de la classe `Compteur`.

D'un point de vue de fonctionnalité, la classe `Compteur2` est plus riche que son parent. Cette dernière est en mesure d'incrémenter et décrémenter le comptage. Pourtant, la classe `Compteur2` ne réalise que la surcharge de l'opérateur `--`. La force de l'héritage rend les classes plus fonctionnelles et plus simples à réaliser. La classe `Compteur2` hérite donc des fonctions membres publiques et protégées de son parent `Compteur`. La classe `Compteur2` augmente la fonctionnalité de `Compteur` en instaurant un opérateur de décrémentation. L'héritage étant un processus unidirectionnel, la classe parente `Compteur` ne peut profiter des avantages de son descendant `Compteur2`.

En pratique, la dérivation de classe exige souvent un traitement spécial dans les constructeurs de la classe dérivée. La classe `Compteur2` possède deux constructeurs dans sa déclaration :

La signature des constructeurs de l'enfant et du parent n'a pas besoin d'être identique !

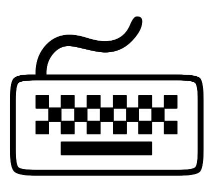
```
Compteur2() : Compteur() { }
Compteur2(int c) : Compteur(c) { }
```

Ces constructeurs sont calqués sur le modèle de la **surclasse** (classe parent de) `Compteur`. D'ailleurs, les constructeurs de `Compteur2` utilisent directement les constructeurs de son parent. La syntaxe:

```
constr_dérivée : constr_parent : { instr }
```

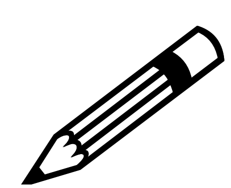
indique au compilateur que le constructeur du parent de la classe dérivée (`constr_parent`) doit être exécuté avant l'exécution des instructions `instr`. Par conséquent, le constructeur du parent est toujours appelé avant celui de l'enfant. Ce qui est tout à fait logique. On doit créer le parent avant de mettre au monde l'enfant.

L'utilisation de toutes ces classes déclarées est fort simple. Il suffit d'inclure les fichiers d'en-tête de `Compteur` et `Compteur2` dans le fichier source du programme.



```
1. /* Fichier Exemple.cpp */
2. #include <iostream.h> // bibliothèque de flux E/S
3. #include "Compteur.h" // en-tête de classe compteur
4. #include "Compteur2.h" // en-tête de classe compteur2
5. void main()
6. {
7. Compteur cpt1;
8. Compteur2 cpt2(5);
9.
10. cout << cpt1.PrendreComptage();
11. cout << cpt2.PrendreComptage();
12.
13. cpt1++; cpt2--;
14. cout << cpt1.PrendreComptage();
15. cout << cpt2.PrendreComptage();
16. ++cpt1; ++cpt2; cpt2++;
17. cout << cpt1.PrendreComptage();
18. cout << cpt2.PrendreComptage();
19. cout << "Bye!"; }
```

Nous avons créé un objet de classe `Compteur` (`cpt1`) et un objet de classe `Compteur2` (`cpt2`). La classe `Compteur2` possède l'opérateur `--` en plus de l'opérateur `++` hérité de son parent et on les a exercé dans les lignes 14 et 18.



Qu'arrive-t-il à la fonction membre `RAZ()` surchargée de la classe `Compteur2` ? La fonction membre `RAZ()` de `Compteur2` possède la même signature et le même type de retour que celle de `Compteur`. Laquelle des deux sera utilisée ?

```
cpt2.RAZ();
```

Puisque l'invocation de `RAZ()` est réalisée à partir d'un objet de la classe dérivée `Compteur2`, c'est la version de cette classe qui sera utilisée. On peut forcer l'invocation d'une fonction membre surchargée de la surclasse par :

```
classe::fnct_membre;
```

Ainsi on aurait pu coder la fonction membre `RAZ()` de `Compteur2` comme suit :

```
void Compteur2::RAZ()
{
Compteur::RAZ();
}
```

La fonction `RAZ()` de `compteur2` invoque celle de son parent.

1.19.1 DÉRIVATION PRIVÉE DES CLASSES

Cette forme de dérivation est plutôt rare. Une classe dérivée par l'utilisation du mot clé `private` est considérée comme une dérivation privée de la surclasse. Une classe dérivée d'une manière privée comporte une restriction supplémentaire. Elle ne peut accéder directement aux données publiques de la surclasse. Voici un exemple.

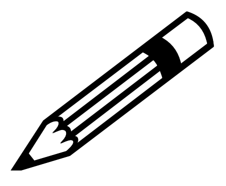
```
class X                 // classe de base
{
public:
  int DataA;
```

```
5. protected:
6.   int DataB;
7. private:
8.   int DataC;
9. };
10.
11.class Y : public X       // dérivation publique
12.{
13.public:
14.  void foo()
15.  {
16.    int a;
17.    a = DataA;           // ok
18.    a = DataB;           // ok
19.    a = DataC;           // Erreur. Interdit !
20.  }
21.}
22.
23.class Z : private X      // dérivation privée
24.{
25.public:
26.  void foo()
27.{
28.    int a;
29.    a = DataA;           // ok
30.    a = DataB;           // ok
31.    a = DataC;           // Erreur. Interdit !
32.  }
33.}
34.
35.void main()
36.{
37.int a;
38.Z objetZ;               // objet de la classe Z
39.a = objetZ.DataA;       // Erreur Interdit !
40.a = objetZ.DataB;       // Erreur Interdit !
41.a = objetZ.DataC;       // Erreur Interdit !
42.}
```

La classe `X` possède trois variables membres : `DataA` (membre public), `DataB` (membre protégé), `DataC` (membre privé). La classe `Y` est une dérivation publique de `X` et la classe `Z` est une dérivation privée de `X`. Les classes `Y` et `Z` possèdent une fonction membre publique appelée `foo()`. On voit dans la fonction `foo()` que l'accès est accepté pour les données membres publiques et protégés. Le comportement des fonctions membres est identique pour la dérivation publique et pour la dérivation privée.

Par contre, un objet d'une classe de dérivation privée ne peut accéder directement à aucune donnée de la surclasse. Les lignes 39 à 41 produisent une erreur de compilation[13]. **Un objet issu d'une classe de dérivation privée n'a aucun accès direct à des données et fonctions de la classe de base. Il peut toujours obtenir les données ou fonctions membres de la classe de base mais seulement au moyen de ses propres fonctions.**

[13] En fait, le compilateur peut générer un ensemble d'erreurs qui indiquent différents types de violation.

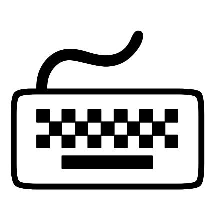


En résumé, une classe dérivée d'une manière privée n'hérite pas directement des capacités de sa surclasse. Elle peut seulement accéder aux données et fonctions (publiques et protégées) de la surclasse par le biais de ses propres fonctions membres. Enfin, la dérivation privée est souvent utilisée dans la construction d'héritage multiple pour éviter l'ambiguïté des parents ayant des fonctions membres de même nom (et même signature)[14].

1.20 FONCTIONS VIRTUELLES

Les fonctions virtuelles permettent l'implantation du comportement particulier des objets (**polymorphisme**). Les fonctions virtuelles sont nécessaires pour donner un comportement particulier à des classes différentes mais dérivées de la même classe de base. Un exemple intuitif est l'affichage graphique des lignes, des rectangles et des cercles. Ces objets sont des objets géométriques mais possèdent des fonctions de traçage distinctes. On peut utiliser une fonction virtuelle `dessine()` pour réaliser le polymorphisme des objets.

On aurait pu penser que le polymorphisme pourrait être réalisé par la surcharge des fonctions. Autrement dit, on surcharge dans les classes les fonctions dont le contenu est différent. Or, la surcharge des fonctions n'est pas une méthode viable. Voici pourquoi.



```
#include <iostream.h>

class ClasseBase
{
public:
  void QuiSuisJe() { cout << "Classe de base\n"; }
};

class ClasseA : public ClasseBase
{
public:
  void QuiSuisJe() { cout << "Classe A\n"; }
};

class ClasseB : public ClasseBase
{
public:
  void QuiSuisJe() { cout << "Classe B\n"; }
};


void main()
{

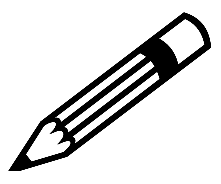
ClasseBase* pBase[2];   // pointeurs type ClasseBase
ClasseA A;
ClasseB B;

// placer les objets dans le tableau de pointeurs
pBase[0] = &A; pBase[1] = &B;

pBase[0]->QuiSuisJe();  // pBase[0] type ClasseA
```

[14] La dérivation privée n'est pas suffisante pour régler le problème d'ambiguïté. Elle ne fait que discipliner le programmeur. On doit aussi utiliser la syntaxe de résolution de l'étendue (*scope*) `classe::fnct_membre` pour enlever toute ambiguïté possible.

```
32.pBase[1]->QuiSuisJe();  // pBase[1] type ClasseB
33.}
```



Nous avons ici trois classes : `ClasseBase`, `ClasseA` et `ClasseB`. Une fonction `QuiSuisJe()` est surchargée dans `ClasseA` et `ClasseB`. À la ligne 29, nous avons assigné les objets de `ClasseA` et `ClasseB` dans un tableau de pointeurs de type `ClasseBase`. **Il est parfaitement légal d'assigner l'adresse d'un objet enfant à un pointeur de type parent**. Ce genre d'assignation est très utilisé puisqu'il suffit d'un pointeur de la classe de base pour entreposer n'importe quel objet des classes dérivées. Sans cette possibilité, on aurait besoin du type exact de l'objet dans le traitement.

Le résultat obtenu de l'exécution du programme est plutôt surprenant :

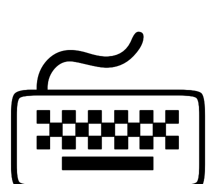
```
Classe de base
Classe de base
```

Les lignes 31 et 33 sont supposées invoquer la fonction surchargée `QuiSuisJe()` de `ClasseA` et `ClasseB`. Mais en réalité, elles invoquent la fonction `QuiSuisJe()` de la classe de base ! On ne pourra pas réaliser le polymorphisme par la surcharge des fonctions. La seule façon d'invoquer les différentes `QuiSuisJe()` est :

```
A.QuiSuisJe();
B.QuiSuisJe();
```

Ce qui n'a pas le même sens que le polymorphisme.

On peut palier à ce problème en ajoutant le mot clé `virtual` devant la fonction `QuiSuisJe()` de la classe de base. Modifions le code de la manière suivante.



```
#include <iostream.h>

class ClasseBase
{
public:
  virtual void QuiSuisJe()
  { cout << "Classe de base\n"; }
};

class ClasseA : public ClasseBase
{
public:
  void QuiSuisJe() { cout << "Classe A\n"; }
};

class ClasseB : public ClasseBase
{
public:
void QuiSuisJe() { cout << "Classe B\n"; }
};


void main()
{
ClasseBase* pBase[2];   // pointeurs type ClasseBase
ClasseA A;
ClasseB B;
// placer les objets dans le tableau de pointeurs
```

```
pBase[0] = &A; pBase[1] = &B;

pBase[0]->QuiSuisJe();  // pBase[0] type ClasseA

pBase[1]->QuiSuisJe();  // pBase[1] type ClasseB
}
```

Redémarrer le programme. Cette fois, l'affichage à l'écran donne le résultat suivant :

```
Classe A
Classe B
```

Le compilateur a donc généré le code nécessaire pour déterminer en-ligne la bonne fonction `QuiSuisJe()` à utiliser. Le polymorphisme est donc réalisé à l'aide du mot clé `virtual`. Les pointeurs `pBase[0]` et `pBase[1]` contiennent les adresses des objets de classes dérivées. La fonction `QuiSuisJe()` est une fonction virtuelle. À son invocation, un mécanisme est enclenché pour déterminer laquelle des fonctions `QuiSuisJe()` utiliser : celle de `ClasseBase`, `ClasseA` ou `ClasseB`. Ce mécanisme porte le nom de liaison tardive (*late-binding*). Contrairement à un appel de fonction ordinaire où l'adresse de la fonction appelée est déterminée lors de la compilation. L'adresse d'une fonction virtuelle n'est déterminée que lorsqu'elle est invoquée. Il existe donc une table de fonction virtuelle et des données supplémentaires identifiant les objets dans l'environnement d'exécution d'un programme C++. Lors de l'invocation d'une fonction virtuelle, le compilateur est en mesure de générer le code pour extraire l'identité de l'objet en question et l'associer à sa fonction virtuelle dans la table virtuelle. Toutes ces opérations sont cachées du programmeur. Évidemment, l'utilisation des fonctions virtuelles (polymorphisme) ajoute de la lourdeur dans l'exécution du programme. Mais c'est le prix à payer pour obtenir cette capacité particulière de la programmation orientée objet.

1.21 POINTEUR THIS

Il existe un pointeur implicite en C++ qui n'a de sens que dans un objet. Ce pointeur implicite est relié à la façon dont le C++ gère ses objets. Dans le C++, chaque objet possède une copie de ses données mais partage une seule instance des fonctions membres[15]. Le pointeur implicite `this` contient l'adresse mémoire de l'objet et le lie à ses fonctions membres. On dit que `this` est une « `auto-référence` » de l'objet puisqu'il se trouve dans un objet et il réfère à l'objet lui-même.

Observer qu'il n'est pas nécessaire de déclarer explicitement le constructeur et le destructeur d'une classe.

```
Classe A {
public:
  A& foo()
  { this->foobar();
    this->iprive = -50;
    return *this;
  }
private:

  void foobar();
```

[15] Un programme exécutable aurait une taille énorme si chacun des objets disposait une copie de ses fonctions membres.

```
  int iprive;
};
```

La classe A possède une fonction membre public `foo()` retournant une référence de classe A. Cette fonction membre exécute une fonction membre privée `foobar()` à l'aide du pointeur implicite `this`. La fonction `foo()` règle également sa variable privée `iprive` au moyen de `this`. Enfin, elle retourne une référence à lui-même par `return *this`.

On aurait pu coder la fonction membre `foo()` de la manière suivante :

```
A& foo()
{ foobar();            // this n'est pas nécessaire
  iprive = -50;        // this n'est pas nécessaire
  return *this;        // this est nécessaire
}
```

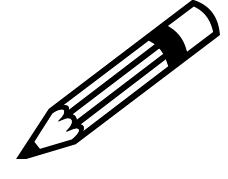
puisque le pointeur `this` est implicite (toujours présent). Cependant, un objet doit utiliser le pointeur `this` lorsqu'il retourne une référence ou pointeur de lui-même.

1.22 Fonctions amies

Une fonction amie permet le relâchement des règles d'accès (masquage de l'information) dans les classes. **Une fonction déclarée amie d'une classe n'est pas membre de cette classe**. Cependant, elle possède le privilège nécessaire pour utiliser les données et fonctions non publiques de la classe. Le mot clé à utiliser est `friend`. Une fonction amie doit apparaître dans le corps de définition d'une classe. À noter qu'une fonction amie n'est pas membre d'aucune classe et elle **n'est pas** modifiée par les mots clés `public`, `protected` ou `private`.

Nous utilisons fréquemment les fonctions amies pour réaliser la surcharge des opérateurs. La raison est que la surcharge des opérateurs n'admet que 0 ou 1 paramètre (voir Dérivation de classes page 32). La syntaxe pour la surcharge des opérateurs est :

```
type operator symbole (param) { instr }
```

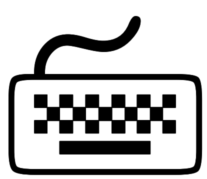


et on ne peut que donner 0 ou 1 paramètre à `param`. On peut facilement contourner ce problème par l'utilisation d'une fonction amie.

L'exemple ci-dessous est donné dans l'aide en-ligne de Visual C++. Dans cet exemple, nous allons créer une classe `Point` capable d'effectuer les additions en surchargeant l'opérateur d'addition:

```
pt = pt + ( expr );
pt = ( expr ) + pt;
```

La première forme implique un objet `Point` à gauche de l'opérateur +. La deuxième forme met en place un objet `Point` à droite de l'opérateur +. On peut réaliser facilement ces surcharges à l'aide des fonctions amies de la classe `Point`.



```
#include <iostream.h>

// déclaration de Point
class Point {
public:
  // Constructeurs
  Point() { _x = _y = 0; }
  Point( unsigned x, unsigned y ) { _x = x; _y = y; }
  unsigned x() { return _x; }
  unsigned y() { return _y; }
  void Print()
  { cout << "Point(" << _x << ", " << _y << ")"
➥<< endl; }

// déclarations des fonctions amies pour la surcharge
// de l'opérateur +
  friend Point operator+( Point& pt, int nOffset );
  friend Point operator+( int nOffset, Point& pt );

private:
    unsigned _x;
    unsigned _y;
};

// définitions des fonctions amies de Point
//
// surcharge de forme Point + int.
Point operator+( Point& pt, int nOffset )
{
Point ptTemp = pt;
    // changer les membres privés _x and _y.
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;

    return ptTemp;
}

// surcharge de forme int + Point.
Point operator+( int nOffset, Point& pt )
{
Point ptTemp = pt;
    // Change private members _x and _y directly.
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;

    return ptTemp;
}

// fonction principale du programme.
void main()
{
    int x = 5;
    int xx = 10;
    Point pt( 10, 20 );
    pt.Print();

    pt = pt + 3;     // Point + int
    pt.Print();

pt = 3 + pt;     // int + Point
pt.Print();

    pt = pt + (x + xx);
    pt.Print();

```

```
66.    pt = (x*2) + pt;
67.    pt.Print();
68.}
```

Le résultat affiché à l'écran est :

```
Point(10,20)
Point(13,23)
Point(16,26)
Point(31,41)
Point(41,51)
```

Lorsque l'expression `pt + 3` est rencontrée dans la fonction `main()`, le compilateur détermine que la fonction amie

```
Point operator+( Point& pt, int nOffset )
```

possède la signature nécessaire. De même pour l'expression `3 + pt`. Le compilateur invoquera la fonction amie

```
Point operator+( int nOffset, Point& pt )
```

Dans ce cas, l'utilisation des fonctions amies est nécessaire pour préserver la propriété de commutativité de l'addition. En effet, la surcharge de l'opérateur + par une fonction membre n'admet qu'un seul paramètre (voir Surcharge des opérateurs à la page 15). Ce qui exclut la réalisation de certaines propriétés de cet opérateur. Une restriction est cependant appliquée pour les surcharges de l'exemple. On doit toujours entourer les expressions de parenthèses (lignes 63 et 66). Sans ces parenthèses, le compilateur émettra des messages d'erreurs lors de la compilateur de la fonction `main()`.

1.23 POINTEURS DE FONCTIONS MEMBRES

Nous avons vu dans la sous-section Pointeurs de fonctions (page 24) qu'un pointeur peut recevoir l'adresse d'une fonction. On peut également utiliser un pointeur pour les fonctions membres d'une classe. Cependant, la syntaxe nécessaire pour réaliser un pointeur de fonction membre est quelque peu différente. D'abord, l'identification d'une fonction membre possède une information supplémentaire. Pour pouvoir identifier correctement une fonction membre, nous avons besoin de :

- sa signature (liste des paramètres d'entrée);
- son type de retour;
- sa classe.

Ces trois éléments sont nécessaires pour déterminer exactement l'identité d'une fonction membre. Ainsi, la fonction membre `int foobar(float x)` de la classe `foo` est déclarée

```
int foo::foobar(float x);
```

et doit être identifiée comme une fonction de la classe `foo` retournant un entier et acceptant un point flottant. Le pointeur recevant cette fonction membre doit être écrit comme suit :

```
int (foo::*pfoobar)(float)
```

où `pfoobar` est le pointeur de la fonction membre `foobar()`. Donc, le type du pointeur est toujours associé à la classe de la fonction membre.

En pratique, nous utilisons souvent le mot clé `typedef` pour simplifier l'écriture des pointeurs de fonctions membres. L'exemple suivant sert à illustrer l'utilisation des pointeurs de fonctions membres.

```
 : : :
class Ecran {
public:
  Ecran& Efface();
  Ecran& NouvelleLigne();
  Ecran& Affiche();
  Ecran& Souligne();
  Ecran& Surbrillance();
  : : :
  : : :
private:
  int x, y;
  char Texte[255];
  : : :
: : :
};


// utiliser typedef pour simplifier l'écriture d'un
// pointeur de fonction membre
typedef Ecran& (Ecran::*Action)();
// une fonction non membre utilisant un pointeur de
// fonction
void Traitement(Ecran& ecran, Action =
↳&Ecran::NouvelleLinge);

void main()
{
Ecran MonEcran;
Action defaut = &Ecran::Affiche;

Traitement(MonEcran);
Traitement(MonEcran, defaut);
Traitement(MonEcran, &Ecran::Souligne);
 : : :
 : : :
}

void Traitement(Ecran& ecran, Action =
↳&Ecran::NouvelleLinge)
{
 : : :
 : : :
code pour le traitement de l'écran
 : : :
 : : :
}
```

La classe `Ecran` représente différents traitements possibles d'un écran de texte. Nous avons défini dans la section publique plusieurs fonctions de traitement de même signature et de même type de retour (`Efface()`, `NouvelleLigne()`, `Affiche()`, `Souligne()`, `Surbrillance()`). Pour simplifier l'écriture des pointeurs de ces fonctions membres, nous avons utiliser le mot clé `typedef` à la ligne 21 :

```
typedef Ecran& (Ecran::*Action)();
```

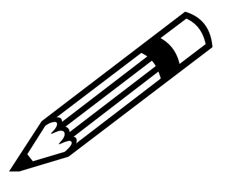
L'identificateur `Action` représentera désormais un pointeur de fonction de type `Ecran`, retournant une référence de type `Ecran` et n'accepte aucun paramètre d'entrée.

Dans la fonction principale `main()`, nous avons défini un pointeur de fonction membre qui représente l'action par défaut de l'écran. Dans notre exemple, l'action par défaut de l'écran est représentée par la fonction membre `Ecran::Affiche()`. Noter bien la syntaxe utilisée dans l'assignation d'une fonction membre à son pointeur. L'opérateur adresse-de (`&`) est nécessaire. On peut aussi passer les fonctions membres comme paramètres dans une fonction non membre[16]. Les lignes 32 et 33 illustrent le passage des fonctions membres dans une autre fonction. On voit que l'utilisation de `typedef` simplifie énormément la lecture et l'écriture du code source.

1.24 MEMBRES STATIQUES

Lorsque appliqué à un membre d'une classe, le mot clé `static` possède une signification plus étendue. En effet, un membre statique (variable ou fonction) agit comme membre global pour la classe seulement. Cela signifie qu'il n'y aura pas de conflit de nom avec les autres variables globales du programme. Un membre statique peut être non public, conservant ainsi l'effet du masquage d'information. Lorsqu'un membre est déclaré statique, il n'existe qu'une seule instance du membre. Par exemple,

```
// fichier foo.h
class foo {
public:
  int couleur, grandeur;
private:
  static float cout;
};
```



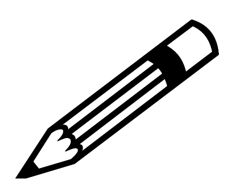
La classe `foo` possède deux variables publiques `couleur` et `grandeur` et une variable privée statique `cout`. Puisque `cout` est un membre statique, nous pouvons l'initialiser **sans avoir** à créer un objet de classe `foo`. De plus, `cout` est statique, il n'y aura qu'une seule copie de cette variable pour tous les objets de classe `foo`. Donc, dans le contexte de son utilisation, nous avons

[16] Une fonction membre peut utiliser une autre fonction membre directement pourvu qu'elles sont de la même classe.

```
1. #include "foo.h"
2.  : : :
3. enum {PETIT, GRAND};
4. enum {NOIR, ROUGE};
5.
6. float foo::cout = 10.99;      // initialisation
7.  : : :
8.  : : :
9. foo foobar1, foobar2 ;        // 2 objets de classe foo
10.  : : :
11. foobar1.couleur = NOIR; foobar1.grandeur = PETIT;
12. foobar2.couleur = ROUGE; foobar2.grandeur = GRAND
13. foobar1.cout = foobar1.cout + 10.0; // cout = 20.99$
14.  : : :
15. foobar2.cout -= 5.00;               // cout = 15.99$
16.  : : :
17. foo::cout = 10.99;                  // cout = 10.99$
18.  : : :
```

La ligne 6 initialise le membre statique `cout`. Noter qu'il n'y a pas encore création d'objet de classe `foo`. On a donné un coût de 10.99 à tous les objets de cette classe. Ce n'est qu'à la ligne 9 que l'on crée deux objets (instances) de classe `foo`. Dès leur création, les objets `foobar1` et `foobar2` ont leur membre privé `cout` initialisé à 10.99. En effet, `foobar1` et `foobar2` partagent la même copie de `cout`. À la ligne 13, l'objet `foobar1` modifie le membre statique à 20.99. Puisque `foobar2` possède la même copie de `cout`, `foobar2` aussi verra le contenu de `cout` monté à 20.99. À la ligne 15, `foobar2` modifie le membre statique `cout` à 15.99. À cet instant, foobar1 aussi aura un `cout` égal à 15.99. Enfin, le programme remet le coût de `foo` à 10.99 (ligne 17). Parce qu'il n'existe qu'une seule copie de `cout` dans la classe `foo`, il est possible de l'invoquer sans faire de référence aux objets.

Donc, l'utilisation d'une variable membre statique permet la communication de l'information à tous les objets d'une même classe.



Une remarque est de rigueur ici. Même si un membre statique permet la transmission de l'information à tous les objets d'une classe, il ne faut croire qu'il est sanctionné d'utiliser des variables globales à toutes les sauces. L'étendue des membres statiques est confinée à l'intérieur d'une classe. Mais on peut quand même écrire du code spaghetti par l'utilisation surabondante des variables membres statiques.

Nous pouvons aussi déclarer une fonction membre par le mot clé `static`. La différence entre une fonction membre non statique et une fonction membre statique est que cette dernière **n'a pas accès au pointeur implicite** `this` **de l'objet**. On ne peut pas utiliser les mots clés `const` ou `volatile` devant une fonction membre statique. L'utilisation principale des fonctions membres statiques est la manipulation des données statiques des classes. La syntaxe d'invocation des fonctions membres statiques est identique à celle des fonctions membres non statiques. Il permet donc une certaine transparence dans leur utilisation. Puisqu'une fonction membre statique manipule des données statiques, elle peut aussi être invoquée avant la définition des objets.

1.25 GESTION DES EXCEPTIONS

Les exceptions sont des conditions anormales rencontrées lors de l'exécution d'un programme. Par exemple, le disque plein, la division par zéro, le manque de mémoire vive, etc. Dans le contexte de C++, ces conditions anormales peuvent être gérées d'une façon simple et structurée. Le principe général de la gestion des exceptions en C++ implique deux acteurs : *i*) une fonction qui envoie un **signal d'exception** vers le programme; *ii*) un bloque de code qui traitement le signal d'exception reçue. Nous allons expliquer plus en détail ce principe de transmission et réception du signal d'exception dans un programme C++.

En pratique, les signaux d'exception sont représentés par des objets

D'abord, un signal d'exception est une entité représentée par des simples types de base (`char`, `int`, `float`, `double`, etc.), par des classes (c'est-à-dire, des classes peuvent jouer le rôle de signal d'exception) et des objets. Donc, un signal d'exception peut contenir un nombre (entier ou en point flottant), une chaîne de caractères. La signification de ce contenu est déterminée par le programmeur. En pratique, nous utiliserons des objets comme signaux d'exception. La raison principale de ce choix pratique est qu'un objet est beaucoup plus flexible qu'un simple entier ou d'une simple chaîne de caractères.

La transmission d'un signal d'exception est la responsabilité des fonctions (membres ou non membres)

En C++, les fonctions sont des unités de travail fondamentales. En effet, dans un programme C++, ce sont toujours des fonctions (membres) qui sont chargées d'exécuter les actions pré-programmées. Il est dont normal que ces dernières soient également responsables d'enclencher les signaux d'exception. Par exemple, la fonction `foo()` a la responsabilité d'enregistrer sur disque les messages reçus de l'Internet. La fonction `foo()` peut reconnaître facilement la situation où elle ne peut écrire les messages reçus par manque d'espace sur le disque. Nous pouvons alors donner à la fonction `foo()` la capacité de transmettre un signal d'exception pour indiquer la condition anormale rencontrée.

Le mot clé `throw` (voir Tableau 2) est utilisé par une fonction pour expédier un signal d'exception vers le programme. Une fonction utilise `throw` lorsqu'elle désire envoyer un signal d'exception.

La réception des signaux d'exception est réalisée grâce à `catch()`.

S'il existe une façon d'expédier un signal d'exception vers le programme, il serait naturel qu'une méthode semblable soit prévue pour recevoir un signal d'exception émanant des fonctions. Le mot clé `catch` (voir Tableau 2) est justement réservé à cet effet. La construction de `catch` est semblable à celle d'une fonction normale mais le mot clé `catch` n'est pas réellement une fonction C++. En effet, `catch` n'a qu'un seul rôle : celui de délimiter le bloc de code qui gère les signaux d'exception reçus. De plus, le bloc de code entouré par `catch` doit nécessairement précédé par un autre bloc de code. Cet autre bloc de code est identifié par le mot clé `try`. Ce dernier est également un mot clé réservé de C++. L'ensemble `try-catch` sert à délimiter les blocs de code qui gèrent les exceptions dans un programme C++. Le schéma ci-dessous donne une idée générale de la construction `try-catch`.

La structure générale des blocs `try-catch`.

```
: : :
: : :
try {
```

```
  : :
// Exécuter les fonctions. Les fonctions entourées par try peuvent générer
// des signaux d'exception
  : :
}
catch (type) {
  : :
// Bloc de code qui gère les exceptions potentielles du bloc try
  : :
}
: : :
: : :
```

Le bloc `try { : : : }` indique au compilateur de mettre en surveillance les fonctions entre accolades. Dès le signalement des exceptions est détecté, les signaux d'exception sont immédiatement acheminés au bloc `catch(type) { : : :}`. C'est dans le bloc de `catch` qu'aura lieu le traitement des exceptions. Donc, le bloc de `try` précède toujours le bloc de `catch`.

À noter que le bloc `catch(type) { : : : }` comporte une capacité de sélection. En effet, le paramètre `type` sert à sélectionner le signal d'exception à traiter. Cette sélection est basée sur le type de données représentant le signal d'exception. Par exemple, `catch (int iex) { : : :}` ne traitera qu'un signal représenté par des entiers (`int`).

Une construction de `catch` en cascade pour traiter différents types de signaux d'exception.

Les fonctions entourées par `try` peuvent générer divers signaux d'exception. Pour traiter différents types de signaux d'exception, nous devons mettre plus d'un bloc de `catch` en cascade. Voici un schéma de cet arrangement.

```
: : :
: : :
try {
  : :
// Par exemple, les fonctions entourées par try peuvent générer des signaux
// d'exception représentés par des int, float et long
  : :
}
catch (int iex) {
  : :
// Bloc de code qui gère les exceptions de type int
  : :
}
catch (float fex) {
  : :
// Bloc de code qui gère les exceptions de type float
  : :
}
catch (long lex) {
  : :
// Bloc de code qui gère les exceptions de type long
  : :
}
: : :
```

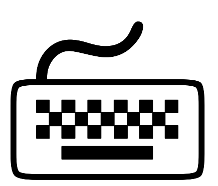
Le code source ci-dessous montre une utilisation simple de la gestion des exceptions. Pour bien comprendre l'exemple suivant, nous avons établi une liste montrant les fonctions, les classes et les objets en jeu.

- `foo()` : une simple fonction C++. Elle a pour rôle d'envoyer un signal d'exception vers le programme. La fonction `foo()` simule donc une fonction capable de détecter une condition anormale pendant son exécution. Le mot clé `throw` est utilisé pour envoyer le signal d'exception `CTest` vers le programme.

- `CTest` : une classe C++. Elle jour le rôle du signal d'exception qui sera utilisé par la fonction `foo()`. Donc, cet exemple utilise une classe et non un objet comme signal d'exception. Ceci est tout à fait légal dans le mécanisme d'exception de C++.

- `CDemo` : une classe C++. Elle sert à montrer l'effet d'une exception sur la vie des objets. C'est pour cette raison que la classe `CDemo` ne contient pas de fonction membres utiles.

- `main()` : fonction principale de ce programme. Les blocs `try-catch` sont instaurés dans cette fonction. Un bloc `try` entoure la fonction `foo()`. Cette dernière est une fonction qui peut envoyer un signal d'exception. Deux blocs de `catch` sont placés en cascade pour recevoir les signaux émanant du bloc de `try`. Le premier `catch` reçoit les signaux de type `CTest`, le deuxième `catch` reçoit les signaux de type `char *`. Évidemment, puisque `foo()` n'utilise que le signal de type `CTest`, le bloc de catch acceptant les signaux de type `char *` ne sera jamais exécuté dans cet exemple.



```
/* $Id: DemoException.cpp,v 1.1.1.1 2000/12/10 17:57:03 twong Exp $
   $Revision: 1.1.1.1 $

  ---------------------------------------------------------
   GPA789 Analyse et conception orientées objet
   Tony Wong, Ph.D., ing.
   Département de génie de production automatisée
  ---------------------------------------------------------
  Note: le code source est formaté avec une tabulation de 2
*/

#include "stdafx.h"
#include <iostream.h>

/* Prototype de la fonction foo()
*/
void foo( void );

/* CTest: classe qui jouera le rôle d'un signal d'exception
    Un objet de cette classe est envoyé par la fonction foo()
  au programme.
*/
class CTest
{
public:
    CTest(){};
    ~CTest(){};
    const char *ShowReason() const {
          return "Exception rencontree dans la classe CTest"; }
};

class CDemo
{
public:
 CDemo() {
  cout << "Construction de CDemo." << endl;
 }
```

```
37.  ~CDemo() {
38.  cout << "Destruction de CDemo." << endl;
39. }
40.};
41.
42.
43./* foo(): fonction qui signalera une exception au programme. On peut imaginer
44.   que la fonction foo() a subi un problème de "disque plein" ou "mémoire
45.   insuffisante".
46.*/
47.void foo()
48.{
49.
50.    CDemo D;
51.    cout<< "Dans la fonction foo(). Signalement d'une exception." << endl;
52.    throw CTest();
53.    cout << "Apres execution de throw." << endl;
54.}
55.
56./* Fonction principale de cet exemple.
57.*/
58.int main()
59.{
60.    cout << "Dans main()." << endl;
61.    try
62.    {
63.        cout << "Dans le bloc de try, Execution de foo()." << endl;
64.        foo();
65.    }
66.    catch( CTest E )
67.    {
68.        cout << "Dans le premier bloc de catch." << endl;
69.        cout << "Exception de type \"CTest\" capturee." << endl;
70.        cout << "Raison connue: " << E.ShowReason() << endl;
71.    }
72.    catch( char *str )
73.    {
74.        cout << "Dans le deuxieme bloc de catch." << endl;
75.        cout << "Exception de type \"char *\" capturee." << endl;
76.        cout << "La chaine est: " << str << endl;
77.    }
78.
79.    cout << "Retour a la fonction principale" << endl;
80.    return 0;
81.
82.}
```

Voici le résultat obtenu après l'exécution de l'exemple :

```
Dans main().
Dans le bloc de try, Execution de foo().
Construction de CDemo.
Dans la fonction foo(). Signalement d'une exception.
Destruction de CDemo.
Dans le premier bloc de catch.
Exception de type "CTest" capturee.
Raison connue: Exception rencontree dans la classe CTest
Retour a la fonction principale
```

Le résultat de cet exemple nous révèle le fait suivant. Lors d'un signalement d'exception, les objets locaux sont automatiquement détruits par le programme. À la ligne 50, un objet de type `CDemo` est créé. Ce dernier affiche à la sortie standard la chaîne « `Construction de CDemo` ». La fonction `foo()` envoie un signal de type `CTest` à la ligne 52. Après quoi, elle affiche à la sortie standard la chaîne « Apres execution de throw ». Or, d'après le résultat obtenu, dès l'exécution de l'instruction

`throw`, les objets créés dans la fonction foo sont détruits. Ce fait est confirmé par la chaîne « `Destruction de CDemo.` » qui indique la destruction de l'objet `D`. De plus, le contrôle du programme est immédiatement passé au bloc de `catch` qui suit le bloc `try`. C'est pour cette raison que la chaîne « `Apres execution de throw.` » de la fonction `foo()` n'est jamais affichée à la sortie standard.

Qu'arrive-t-il à un signalement d'exception sans la gestion par `try-catch` ?

Si pour une raison quelconque, nous avons omis les blocs `try-catch` dans notre fonction principale `main()`, le programme s'exécutera jusqu'au point de l'instruction `throw` et s'arrêtera brusquement par l'affichage d'un message de diagnostic.

```
Dans main().
Dans le bloc de try, Execution de foo().
Construction de CDemo.
Dans la fonction foo(). Signalement d'une exception.
```

Attention! Le message de diagnostic n'est affiché qu'en mode de déverminage.

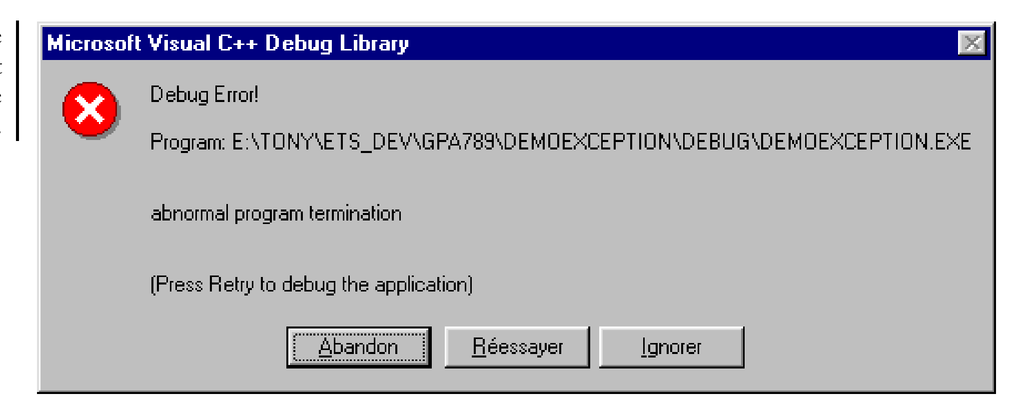


Figure 3 Message de diagnostic. Il n'est disponible qu'en mode de déverminage.

Ainsi, il est toujours nécessaire de gérer les exceptions. Sans quoi, le programme s'arrêtera automatiquement.

Gestion des exceptions d'une manière plus générale

Nous allons modifier le code de notre exemple pour réaliser une gestion plus générale des exceptions. L'invocation du signal d'exception `CTest` est toujours réalisée par la fonction `foo()`. Par contre, la fonction `foo()` est maintenant exécutée par une autre fonction nommée `foobar()`. Donc, le point d'invocation n'est plus directement visible dans le bloc `try`. Voici le schéma montrant cet arrangement.

```
main() ---------------------
  └─► foobar() ---------      Surveillance par try
          └─────► foo() ───► throw CTest
```

Figure 4 Génération du signal d'exception par une sous fonction.

```
/* $Id: DemoException.cpp,v 1.1.1.1 2000/12/10 17:57:03 twong Exp $
   $Revision: 1.1.1.1 $

  -----------------------------------------------------------
   GPA789 Analyse et conception orientées objet

   Tony Wong, Ph.D., ing.
   Département de génie de production automatisée
```

```
 ---------------------------------------------------------
 Note: le code source est formaté avec une tabulation de 2
*/

#include "stdafx.h"
#include <iostream.h>

/* Prototypes des fonctions
*/
void foobar( void );
void foo( void );

/* CTest: classe qui jouera le rôle d'un signal d'exception
   Un objet de cette classe est envoyé par la fonction foo()
 au programme.
*/
class CTest
{
public:
    CTest(){};
    ~CTest(){};
    const char *ShowReason() const {
       return "Exception rencontree dans la classe CTest"; }
};

class CDemo
{
public:
 CDemo() {
 cout << "Construction de CDemo." << endl;
 }
  ~CDemo() {
 cout << "Destruction de CDemo." << endl;
 }
};

/* foobar(): fonction qui appelle une autre fonction. Dans ce cas,
   foobar() appelle foo(). C'est foo() qui signalera une exception.
*/
void foobar()
{
 cout << "Dans foobar(). Execution de la fonction foo()." << endl;
 foo();
}

/* foo(): fonction qui signalera une exception au programme. On peut imaginer
   que la fonction foo() a subi un problème de "disque plein" ou "mémoire
 insuffisante".
*/
void foo()
{

    CDemo D;
    cout<< "Dans la fonction foo(). Signalement d'une exception." << endl;
    throw CTest();
    cout << "Après exécution de throw." << endl;
}

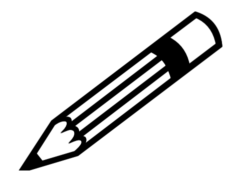
/* Fonction principale de cet exemple.
*/
int main()
{
    cout << "Dans main()." << endl;
try
    {
        cout << "Dans le bloc de try, Execution de foo()." << endl;
        foobar();

    }
    catch( CTest E )
```

```
76.    {
77.        cout << "Dans le premier bloc de catch." << endl;
78.        cout << "Exception de type \"CTest\" capturee." << endl;
79.        cout << "Raison connue: " << E.ShowReason() << endl;
80.    }
81.    catch( char *str )
82.    {
83.        cout << "Dans le deuxieme bloc de catch." << endl;
84.        cout << "Exception de type \"char *\" capturee." << endl;
85.   cout << "La chaine est: " << str << endl;
86.    }
87.
88.    cout << "Retour a la fonction principale" << endl;
89.    return 0;
90.}
```

À la ligne 49, la fonction `foobar()` exécute la fonction `foo()`. Cette dernière va déclencher un signal d'exception à la ligne 62. Le bloc de `try` met la fonction `foobar()` sous surveillance à la ligne 73. Après l'exécution de cet exemple, nous obtenons le résultat suivant :

```
Dans main().
Dans le bloc de try, Execution de foo().
Dans foobar(). Execution de la fonction foo().
Construction de CDemo.
Dans la fonction foo(). Signalement d'une exception.
Destruction de CDemo.
Dans le premier bloc de catch.
Exception de type "CTest" capturee.
Raison connue: Exception rencontree dans la classe CTest
Retour a la fonction principale
```



Le signal d'exception généré par `foo()` est capté par le bloc `catch(CTest)` même si le signal n'est pas généré directement dans le bloc `try`. La Figure 5 illustre le travail du mécanisme d'exception.

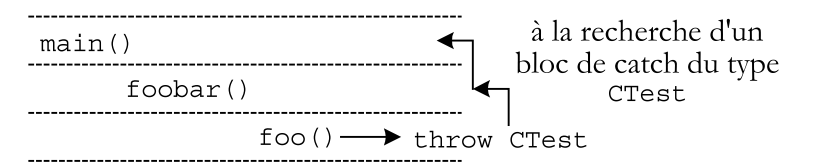


Figure 5 Après l'invocation d'un signal d'exception, le mécanisme d'exception parcoure le programme à la recherche d'un bloc de catch acceptant le type `CTest`.

Après le lancement du signal d'exception par l'instruction `throw`, le mécanisme d'exception[17] tentera de trouver un bloc `catch` de même type que le signal d'exception. La recherche suivra la pile des **appels des fonctions** (*call stack*)[18] en remontant, si nécessaire, jusqu'à la fonction principale du programme (c'est-à-dire, `main()`). Dans notre programme exemple, l'instruction `throw` est exécutée dans la fonction `foo()`. Puisqu'il n'y a pas de catch de type `CTest` dans la fonction `foo()`, le mécanisme d'exception remonte un niveau de la pile des appels.

[17] Le mécanisme d'exception est intégré dans l'environnement d'exécution du programme (*runtime environment*)

[18] Avant l'exécution d'une fonction, l'environnement d'exécution place son adresse dans une pile. De cette façon on peut retracer les appels de fonction effectués dans le programme.

Malheureusement, la fonction `foobar()` ne possède pas de `catch` de type `CTest` et on doit remonter un niveau encore vers la fonction principale `main()`. C'est finalement dans la fonction `main()` que l'on retrouve le catch correspondant. Le mécanisme d'exception exécute alors les instructions du bloc de `catch(CTest)`.

Un exemple concret de l'utilisation des exceptions et de leur gestion.

Voici un deuxième exemple illustrant l'utilisation pratique du mécanisme d'exception dans un programme C++. Dans cet exemple, nous allons créer une classe nommée `RPNCalc1`. Il s'agit d'un calculateur à notation polonaise inverse. La notation polonaise inverse est une façon simplifiée de faire des calculs arithmétiques. Supposons que l'on désire calculer l'expression :

(5.12 + 3.14) * (7.75 – 12.98) / 9.73.

En notation polonaise inverse, cette même expression est réécrite en :

5.12 3.14 + 7.75 12.98 – * 9.73 /.

La notation polonaise inverse ne nécessite pas de parenthèses pour indiquer la préséance des opérations. Les opérandes, dans cette notation, précèdent les opérateurs de sorte qu'une simple pile suffise comme structure de données. La Figure 6 est une explication du fonctionnement d'un tel calculateur.

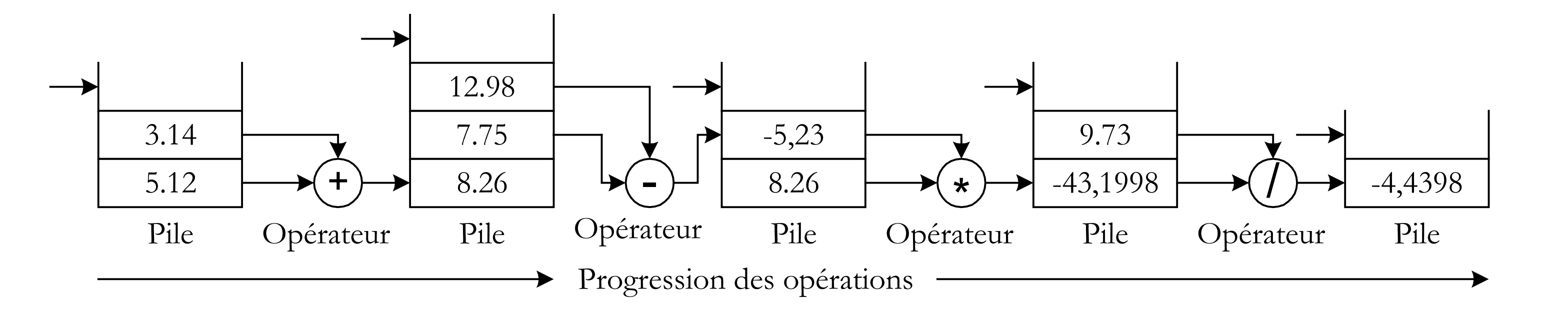


Figure 6 Opérations d'une calculatrice à notation polonaise inverse.

La liste des classes utilisées dans le programme `RPNCalc1` est donnée ci-dessous :

- `RPNException` : une classe qui joue le rôle de signal d'exception pour cet exemple.
- `RPNParser` : une classe qui joue le rôle d'analyseur lexique. Un objet de cette classe est en mesure de reconnaître les éléments d'une expression arithmétique. Les éléments légaux trouvés sont appelés « jetons ».
- `RPNCalc1` : une classe capable de calculer une expression arithmétique selon la notation polonaise inverse. L'algorithme de calcul est montré dans la Figure 6.

L'utilisation de ce programme est effectuée via la ligne de commande DOS. Voici un exemple en supposant que le programme a été placé dans le répertoire `e:\cours\GPA789RPNCalc1` :

```
E:\Cours\GPA789RPNCalc1>rpncalc1 "5.12 3.14 + 7.75 12.98 - * 9.73 /"
Expression: 5.12 3.14 + 7.75 12.98 - * 9.73 /
Resultat: -4.4398561
```

Donc, l'expression en notation polonaise inverse est passée directement au programme via la ligne de commande et **elle doit être placée entre guillemets**.

Le principe de fonction de ce programme est fort simple. Après la réception de l'expression arithmétique, on exécute l'analyseur lexique (`RPNParser`). Ce dernier sépare les jetons et les mémorise d'une manière interne. L'analyseur lexique est ensuite passé dans la calculatrice (`RPNCalc1`) et elle effectue le calcul en demandant à l'analyseur lexique de lui donner les jetons trouvés dans l'expression arithmétique reçue.

Il existe un nombre de situations où des difficultés pourraient survenir. Par exemple, une expression avec des caractères illégaux. Une expression contenant des opérateurs inconnus (le programme n'accepte que : `+`, `-`, `*` et `/`). Une expression avec des opérandes ou des opérateurs manquants. Évidemment, la fameuse division par zéro. Toutes ces conditions anormales doivent être traitées afin d'augmenter la robustesse du programme. L'utilisation des exceptions peut faciliter grandement la gestion de ces situations anormales.

Le fichier en-tête du programme `RPNCalc1` (`RPNCalc1.h`)

```
/*
   GPA789 Analyse et conception orientées objet
  GPA777 Introduction au génie logiciel

  Tony Wong, Ph.D., ing.
  Département de génie de production automatisée
  École de technologie supérieure
  Université du Québec
   (c) Copyleft 2000-2001

    Calculatrice à notation polonaise inverse (modèle #1)
    File: $Source: I:\\CVS_REPOSITORY/GPA789/RPNCalc/RPNCalc1/RPNCalc1.h,v $

  Last modification: $Date: 2000/12/12 01:57:37 $
    Current Revision: $Revision: 1.1.1.1 $

    Id: $Id: RPNCalc1.h,v 1.1.1.1 2000/12/12 01:57:37 twong Exp $

*/
#ifndef _RPNCALC1_H
#define _RPNCALC1_H

// Enlever les avertissement concernant la liste de throw
#pragma warning(disable : 4290)

#include <string.h>                // pour strcpy

const int RPN_EXCEPTION_CHAR_LEN = 256;
const int RPN_CALC_STACK_SIZE = 512;
const int RPN_PARSER_MAX_TOKEN = 512;

const int RPN_PARAM_CHAR_LEN = 128;
typedef enum { NO_TOKEN, ERR_TOKEN, PARAM_TOKEN, ADD_TOKEN, SUB_TOKEN,
 MULT_TOKEN, DIV_TOKEN } TOKEN_TYPE;
```

```
typedef struct _TOKEN {
 double value;
 TOKEN_TYPE token;
} TOKEN;
class RPNException
{
public:
 RPNException(void)
 {
  strcpy(m_Raison, "Pas de raison explicite.");
 }
 RPNException(char *raison)
 {
 strncpy(m_Raison, raison, RPN_EXCEPTION_CHAR_LEN-1);
     m_Raison[RPN_EXCEPTION_CHAR_LEN-1] = '\0';
 }
 char* IndiqueRaison()
 {
  return m_Raison;
 }
private:
 char m_Raison[RPN_EXCEPTION_CHAR_LEN];
};
class RPNParser
{
public:
 RPNParser(void) : m_initialized(false)
 { }
 void InitParser();
 void ParseExpr(char *) throw (RPNException);
 TOKEN* PopToken();
 void PushToken(TOKEN *);
private:
 TOKEN m_Token[RPN_PARSER_MAX_TOKEN+1];
 int m_CurrentToken, m_MaxToken;
 bool m_initialized;
};
class RPNCalc1
{
public:
 RPNCalc1(void) : m_initialized(false)
 { }
 ~RPNCalc1()
 { }
 void RPNCalc1Init()
 {

  memset(m_Pile, 0, sizeof(m_Pile));
  m_top = 0;
  m_initialized = true;
 }
 double Calculate(RPNParser&) throw (RPNException);
private:
 double m_Pile[RPN_CALC_STACK_SIZE];
 int m_top;
 bool m_initialized;

};
 #endif
```

Dans ce programme, deux fonctions peuvent générer des signaux d'exception : *i*) `RPNParser::ParseExpr()` à la ligne 68; *ii*) `RPNCalc1::Calculate()` à la ligne 92. La classe qui joue le rôle du signal d'exception est : `RPNException` déclarée à partir de la ligne 39.

Le code source ci-dessous montre comment les deux fonctions génèrent les signaux d'exception et aussi comment la gestion des exceptions est réalisée.

Le fichier source du programme `RPNCalc1` (`RPNCalc1.cpp`)

```
1./*
2.  GPA789 Analyse et conception orientées objet
3.  GPA777 Introduction au génie logiciel
4.
5.  Tony Wong, Ph.D., ing.
6.  Département de génie de production automatisée
7.  École de technologie supérieure
8.  Université du Québec
9.   (c) Copyleft 2000-2001
10.
11.   Calculatrice à notation polonaise inverse (modèle #1)
12.   File: $Source: I:\\CVS_REPOSITORY/GPA789/RPNCalc/RPNCalc1/RPNCalc1.cpp,v $
13.
14.  Last modification: $Date: 2000/12/12 01:57:37 $
15.   Current Revision: $Revision: 1.1.1.1 $
16.
17.   Id: $Id: RPNCalc1.cpp,v 1.1.1.1 2000/12/12 01:57:37 twong Exp $
18.
19.*/
20.#include "stdafx.h"
21.#include <iostream.h>
22.#include <stdlib.h>                 // pour strtod()
23.#include <errno.h>                  // pour ERANGE
24.#include <math.h>                   // pour fabs()
25.#include "RPNCalc1.h"
26.
27.void Usage(void);                   // Prototype de la fonction
28.bool IsANumber(char *);
29.
30./* Afficher l'usage de ce programme
31.*/
32.void Usage(void)
33.{
34. cout << "<RPNCALC1>" << endl;
35. cout << "RPNCalc1 \"expr\"" << endl;
36. cout << "\"expr\" est une expression en notation polonaise inverse" << endl;
37. cout << "Note: l'expression doit etre entouree de guillemets.\n\n";
38.}
39.
40./* Vérifier si la chaîne représente un nombre
41.*/
42.bool IsANumber(char *number)
43.{
44. char *pStr = number;
45.
46. while (*pStr != '\0')
47.  if (((*pStr < '0') || (*pStr > '9')) && (*pStr != '.')
48.     && (*pStr != '-') && (*pStr != '+'))
49.   return false;
50.  else
51.   pStr++;
52.
53. return true;


54.}
55.
56./* RAZ le parser
```

```
*/
void RPNParser::InitParser()
{
 // 1) RAZ les compteurs de jetons
 m_MaxToken = m_CurrentToken = 0;
 // 2) Créer un jeton vide
 m_Token[RPN_PARSER_MAX_TOKEN].token = NO_TOKEN;
 m_Token[RPN_PARSER_MAX_TOKEN].value = 0;
 m_initialized = true;

}

/* Separer les jetons de la ligne d'entrée
*/
void RPNParser::ParseExpr(char *ligne)
{
 char *pLigne = ligne;
 char localtoken[RPN_PARAM_CHAR_LEN];
 int count = 0, i = 0;
 bool InToken;

 // 1) Il faut que le parser soit initialisé
 if (!m_initialized)
  throw RPNException("Le parser n'est pas pret.");
 // 2) Vérifier si la ligne est vide
 if (strlen(ligne) == 0)
  throw RPNException("La ligne d'entree est vide.");
 // 3) Sauter les espaces au début de la ligne
 while ((*pLigne == ' ') || (*pLigne == '\t')) pLigne++;
 // 4) Chercher les jetons
 while (count <= (signed)strlen(ligne)) {
  if ((*pLigne != ' ') && (*pLigne != '\t') && (*pLigne != '\0')) {
   localtoken[i++] = *pLigne;
   InToken = true;
  }
  else if (InToken) {
   InToken = false;
   // 4.1) Fin d'un jeton
   localtoken[i] = '\0';
   i = 0;
   // 4.2) Voir le type de jeton reçu
   if (!strcmp(localtoken, "+"))
    m_Token[m_MaxToken++].token = ADD_TOKEN;
   else if (!strcmp(localtoken, "-"))
    m_Token[m_MaxToken++].token = SUB_TOKEN;
   else if (!strcmp(localtoken, "*"))
    m_Token[m_MaxToken++].token = MULT_TOKEN;
   else if (!strcmp(localtoken, "/"))
    m_Token[m_MaxToken++].token = DIV_TOKEN;
   else {
    if (!IsANumber(localtoken))
     throw RPNException("Erreur dans l'un des parametres.");
    m_Token[m_MaxToken].value = strtod(localtoken, (char **)(NULL));
    if (fabs(m_Token[m_CurrentToken].value) == HUGE_VAL)
     throw RPNException("Débordement de grandeur dans l'un des parametres.");

    m_Token[m_MaxToken++].token = PARAM_TOKEN;
   }
  }
  pLigne++;
  count++;
 }
 m_initialized = false;
}

/* Obtenir un jeton de la pile

*/
TOKEN* RPNParser::PopToken()
{
```

```
 if (m_CurrentToken < m_MaxToken)
  return &m_Token[m_CurrentToken++];
 else
  return &m_Token[RPN_PARSER_MAX_TOKEN];

}

/* Mettre un jeton dans la pile
*/
void RPNParser::PushToken(TOKEN *token)
{
 if (m_CurrentToken > 0)
  m_Token[--m_CurrentToken] = *token;
}

/* Calculer l'expression en notation polonaise inverse
*/
double RPNCalc1::Calculate(RPNParser& parser)
{
 TOKEN *token;

 // 1) Vérifier l'état
 if (!m_initialized)
  throw RPNException("La calculatrice n'est pas prete.");
 // 2) Chercher le premier jeton
 token = parser.PopToken();
 while (token->token != NO_TOKEN) {
  switch (token->token) {
  case PARAM_TOKEN:
   if (m_top < RPN_CALC_STACK_SIZE)
    m_Pile[m_top++] = token->value;
   else
    throw RPNException("Débordement de la pile calculatrice. Erreur PARAM_TOKEN.");
   break;
  case ADD_TOKEN:
   if (m_top > 1) {
    m_top--;
    m_Pile[m_top-1] += m_Pile[m_top];
   }
   else
    throw RPNException("Erreur ADD_TOKEN.");
   break;
  case SUB_TOKEN:
   if (m_top > 1) {
    m_top--;
    m_Pile[m_top-1] -= m_Pile[m_top];
   }
   else
    throw RPNException("Erreur SUB_TOKEN.");
   break;
  case MULT_TOKEN:
   if (m_top > 1) {
    m_top--;
    m_Pile[m_top-1] *= m_Pile[m_top];
   }
   else
    throw RPNException("Erreur MULT_TOKEN.");
   break;
  case DIV_TOKEN:
   if (m_top > 1) {
    m_top--;
    if (m_Pile[m_top] == 0.0f)
     throw RPNException("Division par zero.");
    m_Pile[m_top-1] /= m_Pile[m_top];
   }

   else
    throw RPNException("Erreur DIV_TOKEN.");
   break;
```

```
194.   default:
195.     throw RPNException("Erreur dans le type de jeton.");
196.   } // fin de switch-case
197.
198.   // 3) Chercher le prochain jeton
199.   token = parser.PopToken();
200.  } // fin de while
201.
202.  // 4) Présenter le résultat ...
203.  if (m_top == 1)
204.   return m_Pile[m_top-1];
205.  else
206.   throw RPNException("Trop d'operandes.");
207.
208. }
209.
210. /* Fonction principale du programme
211. */
212. int main(int argc, char* argv[])
213. {
214.  // 1) Créer un objet de type RPNParser
215.  RPNParser parser;
216.  // 2) Créer un objet de type RPNCalc1
217.  RPNCalc1 calculatrice;
218.
219.  // 3) Vérifier le nombre de paramètres de la ligne de commande
220.  if (argc < 2) {
221.   Usage();
222.   return 1;
223.  }
224.
225.  // 4) Init. le parser et la calculatrice
226.  parser.InitParser();
227.  calculatrice.RPNCalc1Init();
228.
229.  // 5) Analyser l'expression par le parser et
230.  //    évaluer l'expression par la calculatrice
231.  try {
232.   double val;
233.   char result[25];
234.
235.   parser.ParseExpr(argv[1]);
236.   val = calculatrice.Calculate(parser);
237.
238.   cout << "Expression: " << argv[1] << endl;
239.   sprintf(result, "Resultat: %.8g\n", val);
240.   cout << result << endl;
241.  }
242.  catch (RPNException Ex) {
243.   cerr << "Expression: " << argv[1] << endl;
244.   cerr << "Exception: " << Ex.IndiqueRaison() << endl << endl;
245.   return 2;
246.  }
247.
248.  return 0;
249. }
```

Les signaux d'exception sont générés par les fonctions membres :

- `double RPNCalc1::Calculate(RPNParser& parser)` (à la ligne 143)
- `void RPNParser::ParseExpr(char *ligne)` (à la ligne 71)

Observer bien la gestion des signaux dans la fonction principale `main()` à l'aide des blocs de `try` et de `catch`. (les lignes 231-241, 242-246). Le mécanisme d'exception de C++ est un avantage indéniable qui permet la gestion centralisée des exceptions.

Pour exercer le mécanisme d'exception, exécuter le programme `RPNCalc1` avec des expressions erronées.

```
E:\Cours\GPA789RPNCalc1>rpncalc1 "5.12 3.14 + 7.75 12.98 - * 9.73 /a"
Expression: 5.12 3.14 + 7.75 12.98 - * 9.73 /a
Exception: Erreur dans l'un des parametres.


E:\Cours\GPA789RPNCalc1>rpncalc1 "5.12 3.14 + 7.75 12.98 - * 9.73"
Expression: 5.12 3.14 + 7.75 12.98 - * 9.73
Exception: Trop d'operandes.


E:\Cours\GPA789RPNCalc1>rpncalc1 "  5.12  3.14 + 7.75   - * 9.73 /"
Expression:   5.12  3.14 + 7.75   - * 9.73 /
Exception: Erreur MULT_TOKEN.


E:\Cours\GPA789RPNCalc1>rpncalc1 "5.12 3.14 + 7.75 12.98 - * 0.0 /"
Expression: 5.12 3.14 + 7.75 12.98 - * 0 /
Exception: Division par zero.
```

Une dernière remarque. Nous avons doté la classe `RPNException` une chaîne de caractères. Cette chaîne sert à entreposer un texte explicatif donnant la cause de l'exception. Cette utilisation des signaux d'exception facilite grandement la vie des utilisateurs et des concepteurs.

1.26 CONVERSION EXPLICITE DES TYPES

Pourquoi convertir les types de données ? La raison première réside dans le fait que le langage C++ est un langage fortement typé. En effet, on ne peut pas assigner des variables de types différents sans penser aux conséquences. Par exemple,

```
1. float pi = 3.1415;
2. int i;
3. : : :
4. : : :
5. i = pi;    // erreur de conversion
```

À cause de la nature différente des types `float` et `int`, la ligne 5 provoquera une erreur de conversion. En effet, le compilateur refusera l'assignation de la ligne parce que pi est un type `float` et i est un type `int` et la conversion entraînera une perte de précision[1]. Une solution à ce problème est l'utilisation de la conversion explicite (*casting*). La conversion explicite indique au compilateur que la conversion est voulue et elle n'est pas une erreur de programmation.

```
1. float pi = 3.1415;
2. int i;
3. : : :

4. : : :
5. i = int(pi);   // conversion explicite de float à int
```

[1] On peut, grâce aux options du compilateur, ignorer ce genre d'erreur. Cependant, il est fortement déconseillé de le faire.

Dans l'exemple ci-dessus, la ligne est une conversion explicite d'une variable de type `float` à une variable de type `int`. Évidemment, le sens (ou la sémantique) de cette conversion doit être géré par le programmeur.

La conversion explicite impliquant des types de base est très simple à utiliser. Il suffit d'entourer la variable à convertir par le mot clé représentant le type à obtenir (i.e. `int(pi)`, `signed(pi)`, `unsigned(pi)`, `long(pi)`, `float(pi)`, `double(pi)`, etc.). La situation est plus nébuleuse lorsque la conversion implique des objets. En effet, à cause de la possibilité de l'héritage des classes, il est parfois difficile de convertir correctement les objets. De plus, dans le cas des objets, une mauvaise conversion provoque fréquemment une erreur d'exécution du programme. En voici un exemple,

Un exemple de conversion explicite erronée impliquant des objets.

```
1. class foo1
2. {
3. public:
4.   fnct1();
5. };
6. class foo2 : public foo1
7. {
8.   fnct2();
9. }
10. : : :
11. : : :
12. foo1 f1;
13. foo2 f2;
14. : : :
15. : : :
16. f1 = foo1(f2);   // conversion explicite
17. f1.fcnt2();      // erreur fonction fnct2() n'est pas membre de la classe foo1
18. f2 = foo2(f1);   // erreur ne peut conversion du type parent vers un type
19.                  // enfant
```

Dans cet exemple, la classe `foo1` est le parent (surclasse) de la classe `foo2`. Les lignes 12 et 13 créent les objets `f1` et `f2` de type `foo1` et `foo2` respectivement. Nous appliquons la conversion explicite de `foo2` à `foo1` à la ligne 16. Malgré la conversion explicite, la ligne 17 génère une erreur de compilation. La raison est que l'objet `f1` de type `foo1` ne possède pas la fonction membre `fnct2()` mais après avoir subi la conversion explicite. La ligne 18 provoque également une erreur de compilation. En effet, le langage C++ ne permet pas la conversion d'un type parent (`foo1`) vers un type enfant (`foo2`).

On pourrait penser que l'utilisation des pointeurs peut surmonter cette difficulté. Voici un exemple.

Un exemple de conversion explicite erronée utilisant un pointeur d'objet.

```
class foo1
{
public:
  fnct1();
};

class foo2 : public foo1
{
  fnct2();

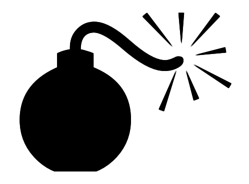
}
: : :
: : :
```

```
12.foo1 *f1;        // un pointeur de type foo1
13.foo2 f2;
14.: : :
15.: : :
16.f1 = (foo1*)&f2; // conversion explicite par pointeur
17.f1->fcnt2();     // erreur fonction fnct2() n'est pas membre de foo1
```

À la ligne 12, un pointeur de type `foo1` est créé. La conversion explicite par pointeur de type `foo2` à `foo1` est réalisée à la ligne 16. Encore une fois, la compilation de la ligne produira une erreur. Donc, même si on utilise un pointeur, cela n'échappe pas à la vigilance du compilateur.

Par contre, il existe un cas dangereux que le compilateur ne peut prévoir. Voici ce cas :

```
1.foo1 *f1;        // un pointeur de type foo1
2.foo2 f2;
3.: : :
4.: : :
5.f1 = (foo1*)&f2; // conversion explicite par pointeur
6.f1->fcnt1();     // pas d'erreur de compilation mais très dangereux
```



Nous avons effectué la conversion explicite par pointeur d'un objet de type `foo2` en un pointeur d'objet de type `foo1`. À la ligne 6, nous avons exécuté la fonction `fcnt1()` de la classe `foo1` via le pointeur `f1`. **Pourtant, jamais un objet réellement de type `foo1` n'a été créé dans le code !** En effet le compilateur a créé pour le programme, d'une manière silencieuse, un objet temporaire de type `foo1`. Cet objet temporaire n'a qu'une existence éphémère. Il sera détruit à la fin de la visibilité locale.

Le problème des objets temporaires est justement leur durée de vie. Puisqu'il est détruit automatiquement par le compilateur, le code peut sembler bien fonctionner jusqu'au moment où la visibilité de l'objet temporaire disparaît. Ce genre d'erreur est très difficile à décerner.

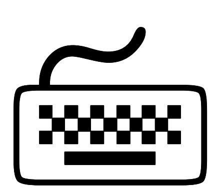
Malgré tous ces problèmes, il arrive que la conversion explicite des objets soit nécessaire. Prenons l'exemple d'une hiérarchie de classes. Plus précisément, un hiérarchique représentant deux types de livre : *i*) référence académique; *ii*) roman fiction. Ces livres ont un point en commun, ils possèdent nécessairement un titre. Par contre, les bouquins académiques ont un attribut supplémentaire : le niveau de scolarité visé.

Ce niveau de scolarité est indiqué par un entier (par exemple, 0 signifie le niveau maternel, 1 signifie le niveau primaire, etc.). Pour rassembler ces types de livres en une seule hiérarchie, nous créons une classe de base appelée `Livre`. Cette classe de base contiendra le titre du livre et une fonction membre `ImprimeTitre()` pour afficher le titre du livre à la sortie standard. Les références académiques sont représentées par la classe `LivreAcademique`. Les romans de fiction sont représentés par la classe `RomanSavon`. Une fonction C++ (non membre) `ImprimeInfo(Livre *)` est utilisée pour afficher l'information des livres à partir d'un pointeur de la classe de base `Livre`.

Le problème potentiel dans cet exemple est que la classe `LivreAcademique` possède un attribut (niveau de scolarité du livre) qui n'est pas dans le reste de la hiérarchie. Une façon simple **mais interdite** consiste à effectuer une conversion explicite vers le type `LivreAcademique` dans la fonction non membre `ImprimeInfo()`. Puis, exécuter la fonction désirée `LivreAcademique::ImprimeNiveau()`. Or, comme l'avons vu, la conversion explicite d'un pointeur d'objet parent en un pointeur d'objet enfant est interdit en C++.

Opérateur de conversion dédiée `dynamic_cast`.

La solution à ce genre de problème est l'utilisation des opérateurs de conversion dédiée de C++. En particulier, dans le cas de cet exemple, l'opérateur `dynamic_cast` permet la conversion dédiée d'un pointeur de type parent en un pointeur de type enfant d'une même hiérarchie d'objets. Le code ci-dessous donne le programme réalisant cette hiérarchie et la solution utilisant la conversion dédiée `dynamic_cast`.



```
// test.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream.h>
#include <string.h>

const int CHAR_LEN = 256;
/* Prototype
*/
class Livre;
void ImprimeInfo(Livre *);

/* La classe de base
*/
class Livre {
public:
 Livre(char *titre) { strcpy(m_Titre, titre); }
 /* Fonction ImprimeTitre() est virtuelle pour permettre
    la réalisation du polymorphisme.
 */
 virtual void ImprimeTitre() const {
  cout << "Titre: " << m_Titre << endl; }


private:
 /* Cacher le constructeur par défaut dans la section
    privée. Une façon de forcer l'utilisation du constructeur
   dans la section publique.
 */
 Livre();
 char m_Titre[CHAR_LEN];

};


/* Référence académique dérivée de la classe Livre
*/
class LivreAcademique : public Livre {

public:
 LivreAcademique(char *titre, int niveau)
  : Livre(titre), m_Niveau(niveau)

 { }
 void ImprimeTitre() const {

  cout << "LivreAcademique ";
  // Imprimer le titre par la classe de base
```

```
 Livre::ImprimeTitre();
}
void ImprimeNiveau() const {
 cout << "Niveau academique: " << m_Niveau << endl; }

private:
/* Cacher le constructeur par défaut dans la section
   privée. Une façon de forcer l'utilisation du constructeur
  dans la section publique.
*/
LivreAcademique();
int m_Niveau;

};


class RomanSavon : public Livre {
public:
RomanSavon(char *titre)
 : Livre(titre)
{ }
void ImprimeTitre() const {
 cout << "Roman Savon: ";
 Livre::ImprimeTitre();
}

private:
/* Cacher le constructeur par défaut dans la section
   privée. Une façon de forcer l'utilisation du constructeur
  dans la section publique.
*/
RomanSavon();
};


void ImprimeInfo(Livre *pLivre)
{
// Imprimer le titre
pLivre->ImprimeTitre();
// S'il s'agit d'un livre académique
LivreAcademique *pLA = dynamic_cast<LivreAcademique *>(pLivre);
// Si c'est réellement un livre académique ...
if ( pLA)
 pLA->ImprimeNiveau();

}


int main(int argc, char* argv[])
{
Livre *pL;
int niveau;
char titre[CHAR_LEN];
int reponse = 1;

while (reponse != 0) {
  /* Obtenir de l'info de l'utilisateur
   */
   cout << "Donner le titre du livre: ";
   cin >> titre;

   do {
    cout << "Donner le Type (1=academique, 2=fiction): ";
    cin >> reponse;

  } while (reponse < 1 || reponse > 2);

   switch (reponse) {
    case 1: cout << "Donner le niveau (academique): ";
```

```
109.              cin >> niveau;
110.                   // Créer un objet de type LivreAcademique
111.           pL = new LivreAcademique(titre, niveau);
112.             break;
113.    case 2: // Créer un objet de type RomanSavon
114.          pL = new RomanSavon(titre);
115.                  break;
116.  }
117.   /* Imprime l'info du livre
118.   */
119.   ImprimeInfo(pL);
120.   cout << "Continuer (0=non)?: ";
121.   cin >> reponse;
122.  }
123.  return 0;
124. }
```

Observez bien le code des lignes 80 à 90 (la fonction non membre `ImprimeInfo()`). Cette fonction accepte un pointeur de type `Livre`. Pour pouvoir imprimer correctement le niveau de scolarité des bouquins académiques, nous avons utilisé la conversion dédiée `dynamic_cast` de la manière suivante :

```
LivreAcademique *pLA = dynamic_cast<LivreAcademique *>(pLivre);
```

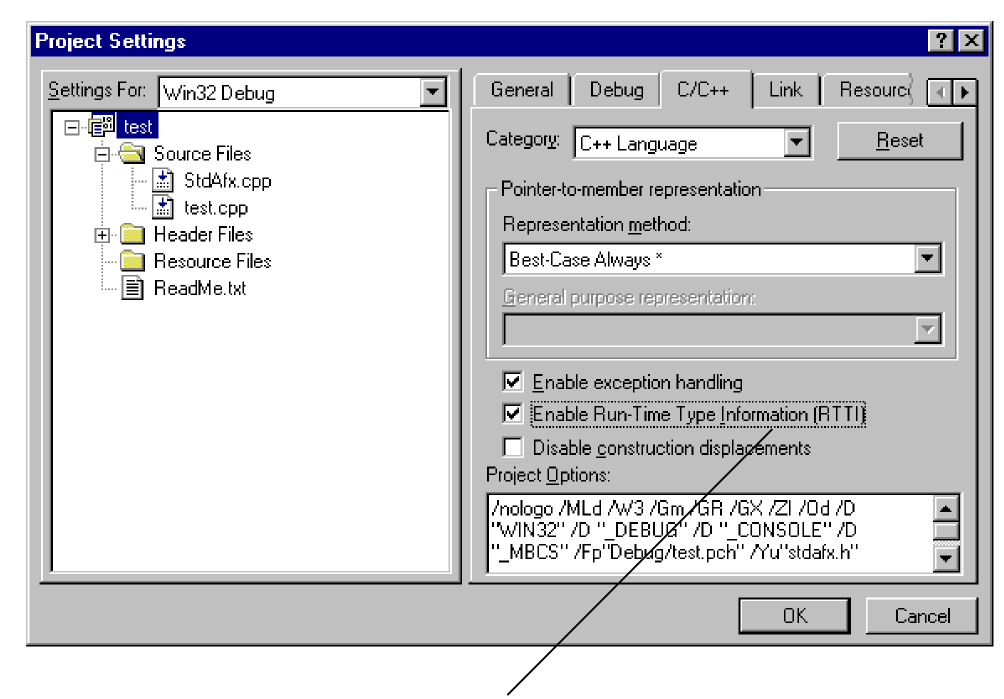
La syntaxe de cette conversion est correcte si les classes `LivreAcademique` et `Livre` sont dans la même hiérarchie de classes. Autrement, `dynamic_cast` retourne un pointeur nul.

Donc, il faut toujours valider le pointeur retourné par `dynamic_cast` (comme à la ligne 87). Encore une fois, la conversion n'a de sens que si elle est légale. C'est la responsabilité du programmeur de s'assurer que l'exactitude de la sémantique. Par contre, l'opérateur `dynamic_cast` est une conversion « sûre » (*type safe*) dans le sens que l'on peut tester la validité du résultat de la conversion.

Enfin, la conversion dédiée `dynamic_cast` utilise une technique spéciale du C++ appelée le RTTI (*RunTime Type Information*). Donc, pour que dynamic_cast effectue correctement son travail, il est nécessaire d'activer l'option RTTI de votre compilateur[2]. Dans l'environnement VC++, l'option RTTI est activée par :

Le panneau des paramètres du projet. C'est dans ce panneau que l'on active l'option RTTI.

[2] Normalement cette option n'est pas activée afin de générer un code plus compact.



Activer le RTTI avant d'utiliser la conversion `dynamic_cast`

La technique et la capacité de RTTI dépassent le cadre de ce cours et elles ne seront pas présentées dans ce document.

Opérateur de conversion dédiée `static_cast`.

Il existe également un opérateur de conversion dédiée appelé `static_cast`. Le résultat d'une conversion de type par `static_cast` est identique à celui d'une conversion explicite de C++. Il n'y a pas de mécanisme de validation par RTTI. Donc, static_cast est une conversion qualifiée de « non sûre » (*type unsafe*). Ainsi, on aurait pu remplacer la ligne 85 de l'exemple précédent par :

```
LivreAcademique *pLA = static_cast<LivreAcademique *>(pLivre);
```

L'opérateur `static_cast` forcera pLivre à prendre le type pointeur de `LivreAcademique` peu importe la sémantique de cette conversion. Le résultat est le suivant :

Résultat de l'exemple lorsque `dynamic_cast` est remplacé par `static_cast`.

```
Donner le titre du livre: Ça
Donner le Type (1=academique, 2=fiction): 2
Roman Savon: Titre: Ça
Niveau academique: -33686019
Continuer (0=non)?:
```

Le livre porte le nom de « Ça »[3] et est un roman de fiction. Or, puisque static_cast retourne toujours un pointeur peu importe la légalité de cette conversion, la logique du programme fait en sorte qu'il affiche quand même le niveau académique! Donc, utiliser l'opérateur `static_cast` avec prudence.

[3] Un roman de Stephan King.

Opérateur de conversion dédiée `const_cast`.

Nous avons vu que le mot clé `const` sert à indiquer la nature constante des données, objets et fonctions. Lorsque appliqué à une fonction, le mot clé `const` signifie que la fonction ne modifie pas l'état de l'objet. Ainsi,

```
void LivreAcademique::ImprimeTitre() const;
```

signifie que `ImprimeTitre()` ne modifie pas les attributs (variables membres) de l'objet qui possède cette fonction membre.

Aussi, le mot clé `const` peut être placé devant un argument de fonction. Dans ce cas, on signale au compilateur que l'argument ne sera pas modifié par la fonction. Prenons l'exemple de la fonction suivante:

```
bool ImprimeTableau(const float *pTab, int l);
```

L'argument `const float *pTab` indique au compilateur que le pointeur `pTab` ne sera pas modifié par la fonction `ImprimeTableau()`. Le code C++ de la fonction `ImprimeTableau()` **ne peut contenir** des instructions d'assignation au pointeur `pTab` (ex : `PTab = &val;`) à cause du mot clé const utilisé dans la liste d'arguments.

Il est possible d'éliminer la nature constante d'une variable ou d'un argument à l'aide de l'opérateur de conversion dédiée `const_cast`. Voici un exemple :

```
1. bool ImprimeTableau(const float *pTab, int l)
2. {
3.   float *pMonPointeur = pTab;  // Erreur! pTab est const
4.   float *pMP = const_cast<float *>(pTab); // solution
5.   for (i=0; i<l; i++)
6.     cout << *(pMP++) << end;
7. }
```

La ligne 3 est refusée par le compilateur puisqu'une variable `const` doit toujours assignée à une autre variable `const`. En utilisant la conversion dédiée `const_cast`, nous avons éliminé la nature constante de la variable (`pTab` dans l'exemple). On peut même changer la valeur de `pTab` après la conversion par `const_cast` mais le résultat est indéterminé et très dangereux pour la logique du programme.

Opérateur de conversion dédiée `reinterpret_cast`.

L'opérateur `reinterpret_cast` est sans doute l'opérateur de conversion dédiée le plus puissant du langage C++. Il permet la conversion d'un type pointeur en n'importe quel autre type de pointeur. Il permet la conversion d'un type intégral (nombres entiers, caractères) en un pointeur et vice versa. En voici un exemple :

```
: : :
typedef unsigned char UCHAR;
float fVal = 3.1415;
// Utiliser reinterpret_cast pour convertir l'adresse d'un float en un
// pointeur de char !
UCHAR *pChar = reinterpret_cast<UCHAR *>(&f);
for (int i=0; i<sizeof(float); i++)
  cout << static_cast<int>(pChar[i]) << endl;
: : :
```

Appliqué à des objets, l'opérateur `reinterpret_cast` permet la conversion d'objets entre différentes hiérarchies. Il est clair que `reinterpret_cast` est

l'opérateur de conversion le plus puissant mais il est également le plus dangereux des opérateurs de conversion.

LECTURE SUGGÉRÉE

Ce chapitre n'est qu'une introduction de quelques éléments importants de C++. Nous avons dû laissé de côté certains sujets avancés comme les types paramétrisés (*template*), les classes collections, le RTTI (*Run-Time Type Information*). Le plus important encore, l'utilisation de la bibliothèque STL (*Standard Template Library*) qui garantit une portabilité parfaite des classes collections. Les types paramétrisés ainsi que le STL seront l'objet d'étude du prochain chapitre.

Les références ci-dessous renferment toutes les informations utiles pour la programmation en C++. Ces références ont été utilisées pour la rédaction de ce chapitre.

Laforce, Robert, *Object-Oriented Programming in C++*. Waite Group Press, 1995.

Ce livre traite d'une manière pratique les éléments du langage C++. Recommandé pour les débutants.

Lippman, Stanley, *C++ Primer*. Addison-Wesley, 1991.

Lippman donne une présentation plus formelle des éléments de C++. Recommandé pour les utilisateurs de niveau intermédiaire.

Ellis, Margaret; Stroustrup, Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

Stroustrup, Bjarne, *The Design and Eveolution of C++*, Addison-Wesley, 1994.

Stroustrup est le concepteur du langage C++. Ce livre est hautement technique et est recommandé pour les programmeurs de niveau avancé.

Note sur la citation parue au début de ce chapitre : Cette citation sert à illustrer pourquoi on confond souvent une fonction mathématique avec un programme qui la calcule. Les fonctions partielles récursives sont des fonctions élémentaires calculables (au sens de la théorie de calculabilité). Donc, une forme de programme. Par contre, il n'y a pas de correspondance 1 à 1 entre une fonction partielle récursive et son programme de calcul. En effet, il peut exister une infinité de programmes qui calculent la même fonction partielle récursive. Dans la citation, ϕ_i représente une fonction partielle récursive et N est l'ensemble des nombres naturels. Ainsi, ϕ_i est infiniment dénombrable. La numérotation de Gödel est utilisée pour encoder une machine de Turing. Puisque nous pouvons simuler une machine de Turing par une fonction partielle récursive, cette dernière est équivalente à une numérotation de Gödel.

PROBLÈMES

**** 1.1 Expliquer la visibilité des variables et fonctions d'un programme C++.

*** 1.2 Pourquoi le type de retour n'est pas un facteur déterminant dans la surcharge des fonctions?.

**** 1.3 Comment peut-on allouer dynamiquement un tableau à *n* dimensions où $n > 2$?

* 1.4 Quelle est la différence entre une classe et un objet dans le langage C++?

1.5 Écrire la définition d'une classe de base représentant un nombre muni d'opérateurs `+`, `-`, `/` et `*`. (voir question 1.6 et 1.7)

1.6 Dériver les classes représentant *N* (nombre naturel), Z (nombre entier) et Q (nombre rationnel) à partir de la classe de base obtenue en 1.5.

1.7 Dériver les classes représentant *R* (nombre réel), *I* (nombre imaginaire) à partir de la classe de base obtenue en 1.5.

1.8 Écrire un programme C++ réalisant une calculatrice RPN (notation polonaise inverse) acceptant des nombres imaginaires. Les opérateurs de cette calculatrice sont {+, -, /, *}.

1.9 Reprendre le programme réalisé en 1.8. Ajouter la gestion d'exception `try-throw-catch` pour gérer la division par zéro et les erreurs d'expression RPN.

*** 1.10 Donner une situation d'utilisation pour chacun des opérateurs de conversion suivants :

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

CHAPITRE 2

Introduction à STL

Un philosophe contemporain, argumentateur à outrance, auquel on reprochait que ses raisonnements irréprochablement déduits avaient l'expérience contre eux, mit fin à la discussion par cette simple parole : « L'expérience a tort ».

— *Henri Bergson, Le rire.*

Le STL est une bibliothèque de structures de données, d'algorithmes, et d'opérations. Elle démontre bien les capacités intrinsèques du langage C++. L'étude de cette bibliothèque nous permettra de bien connaître la puissance de l'approche orientée objet. Les classes contenues dans cette bibliothèque sont avant tout des outils pour le programmeur. Elle sert donc à faciliter la traduction des idées en code source. L'idée de base du STL est de construire des classes et fonctions applicables à tous les types de données. C'est pour cette raison qu'il existe une utilisation abondante de type paramétrisé (*template*). Afin de faciliter l'utilisation des structures de données et les algorithmes intégrés, un nombre de classes de support ont été créées. Ces classes de support sont les **itérateurs**, les **fonctionnels** et les **adaptateurs**. Ces classes sont particulières dans le STL mais leurs concepts peuvent être appliqués à tous projets C++.

2. TYPES PARAMÉTRISÉS

Par exemple, on veut réaliser :

$y = (2ab)/(4+b)$,

si a et b sont des entiers (`int`)

```
int foo(int a, int b)
{
  return ((2*a*b)/(4+b));
}
```

si a et b sont des nombres doubles précisions (`double`)

```
double foo(double a, double b)
{
  return ((2*a*b)/(4+b));
}
```

si *a* et *b* sont des nombres entiers non signés (`unsigned int`)

```
unsigned int foo(unsigned int a, unsigned int b)
{
  return ((2*a*b)/(4+b));
}
```

Dans ce cas, la surcharge des fonctions n'est pas une solution. En effet avec la surcharge des fonctions, nous devons réécrire plusieurs fois la même fonction `foo()` pour les différents types de `a` et `b`. Par contre, on peut utiliser une fonction aux types paramétrisés (*Function template*). La syntaxe d'une fonction aux types paramétrisés est donnée ci-dessous :

```
template <class Type>
Type foo(Type a, Type b)
{
  return ((2*a*b)/(4+b));
}
```

Lorsque le compilateur rencontre l'appel de fonction

```
int y = foo(8,6);              // 12
```

Le compilateur reconnaît qu'il s'agit d'une fonction aux types paramétrisés et crée une instance de la une fonction `foo()` de type `int` (*template function*). Voici un exemple de codage.

```
// Exemple de template (1)
#include <iostream.h>

template <class Type>
Type foo(Type a, Type b)
{
  return ((2*a*b)/(4+b));
}

int main()
{
 int iy = foo(8,6);  // 9
 cout << iy << endl;
 float fy = foo(8.0f, 6.0f); // 9.6
 cout << fy << endl;
 char cy = foo('f', 'b');    // -
 cout << cy << endl;
 return (0);

}
```

Le résultat affiché est :

```
9
9.6
-
```

On peut également créer des fonctions avec plus d'un type paramétrisé. L'exemple ci-dessous montre une fonction

Cet exemple utilise deux fichiers : `template2.h` et `template2.cpp`

```
// Fichier template2.h
#if !defined(TEMPLATE2_H)
#define TEMPLATE2_H

// T1 - int, long, float, double
// T2 - int, long
template <class T1, class T2>
T2 Trouve(T1 *vec, T1 valeur, T2 longueur)
{
 T2 j;
 for (j=T2(0); j<longueur; j++)
  if (vec[j] == valeur)
   return (j);
 return T2(-1);

}

#endif

// Fichier source
#include <iostream.h>
#include "template2.h"

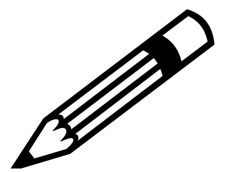
int main()
{
 char cVec[] = {'a', 'b', 'c', 'd', 'e', 'f'};
 char ch = 'b';
 int position = Trouve(cVec, ch, 6);
 cout << "Element " << ch << " trouve a la position: " << position
      << endl;
 double dVec[] = {1.0, 3.0, 5.0, 7.0, 9.0, 11.0};
 double d = 5.0;
 cout << "Element " << d << " trouve a la position: " <<
       Trouve(dVec, d, 6) << endl;

 return (0);

}
```

Le résultat affiché est :

```
Element b trouve a la position: 1
Element 5 trouve a la position: 2
```



Les types paramétrisés `T1` et `T2` peuvent être des types prédéfinis (ex : `int`, `long`, `float`, `double`, etc.). Les types paramétrisés peuvent aussi être des objets de classes différentes. Si les types paramétrisés sont des objets C++, quelles sont les fonctions membres obligatoires que l'on doit réaliser dans l'exemple `template2` ?

- ❑ Opérateur d'assignation

`operator==()`

Voici un exemple qui explique pourquoi ne pas réaliser les types paramétrisés par des macros du langage C++.

```
#include <iostream.h>

#define max(x, y) \
        (((x) > (y)) ? (x) : (y))
```

```
int main()
{
 float a = 12.0f, b = 53.1f;

 float z = max(a,b);
cout << "max de " << a << " et " << b <<
     " est " << z << endl;

 return (0);

}
```

Le résultat affiché est :

```
max de 12 et 53.1 est 53.1
```

Problèmes dans l'utilisation des macros ☞

Problème #1 : Aucune vérification de type par le compilateur. Donc, on peut écrire :

```
#include <iostream.h>

#define max(x, y) \
        (((x) > (y)) ? (x) : (y))


int main()
{
 float a = 12.0f, b = 53.1f;
 char aa = 'g';

 float z = max(a,aa);        // cette ligne compile !
cout << "max de " << a << " et " << b <<
     " est " << z << endl;

 return (0);

}
```

Problème #2 : Effets secondaire subtiles

```
#include <iostream.h>

#define max(x, y) \
        (((x) > (y)) ? (x) : (y))


int main()
{
 // Erreurs subtiles
 int i = 1, j = 2;
 int zz = max(++i, ++j);     // on s'attend à z = 3
 cout << "zz = " << zz << endl;

 return (0);

}
```

Le résultat affiché est :

```
zz = 4
```

Problème #3 : N'oublie pas les parenthèses !

```
#include <iostream.h>
#include <math.h>

#define racine(a,b,c) \
       ((-b+sqrt(b*b-4*a*c))/(2*a))

int main()
{
 // Macro sans parentheses
 float fa=1.0f, fb=1.0f, fc=1.0f;
 cout << "Racine de: x2 + 4x + 1 est " << // -3.7321
  racine(fa, fb+3.0f, fc) << endl;

 return (0);

}
```

Le résultat affiché est :

```
Racine de: x2 + 4x + 1 est 1.86603
```

Ainsi, en C++, on évite d'utiliser les macros. À leur place, il est recommandé d'utiliser les types paramétrisés pour les classes C++. En voici un exemple,

```
#include <iostream.h>
#include <math.h>

template <class Type>
class DeuxXY {
public:
 DeuxXY(Type x1, Type y1)
  : x(x1), y(y1)
 { }
 void Montrer();
private:
 Type x, y;
};

template <class Type>
void DeuxXY<Type>::Montrer()
{
 cout << x / y << endl;
}

int main()
{
 DeuxXY<int> iVar(36,6);
 DeuxXY<double> dVar(sqrt(2.0), 2.0);

 iVar.Montrer();     // 6
 dVar.Montrer();     // .707

 return (0);
}
```

Le résultat affiché est :

```
6
0.707107
```

Cependant, il faut faire attention à la syntaxe des fonctions membres définies en dehors de la déclaration :

```
template <class Type>
void DeuxXY<Type>::Montrer()
{
 cout << x / y << endl;
}
```

Dans l'exemple ci-dessus `DeuxXY<Type>` est l'unité représentant la classe, ce qui mène à une question importante : Si une fonction membre retourne un type de sa classe, quelle sera la syntaxe à utiliser ?

```
template <class T>
class foo {
public :
 foo<T> foobar();
 : :
 : :
};
```

Une seconde question : Si cette fonction membre est définie en dehors de sa déclaration, quelle sera sa syntaxe?

```
template <class T>
class foo {
public :
 foo<T> foobar();
 : :
 : :
};

foo<T> foo<T>::foobar()
{
 : :
 : :
}
```

Pour conclure cette sous-section, nous allons donner un exemple d'application. Cet exemple est la réalisation d'une liste chaînée qui peut accepter tous les types de données. Le codage nécessaire est présenté ci-dessous.

```
// ListeChainee.h
#if !defined(LISTE_CHAINEE_H)
#define LISTE_CHAINEE_H

template <class T>
struct Noeud {
 T donnee;
 Noeud* Prochain;
};

template <class T>
class Liste {
private:
 Noeud<T>* tete;
public:
 Liste() { tete = 0; }       // constructeur
 ~Liste();
 void Ajoute(T data);
 void Affiche();

 bool Enleve(T data);
};
#endif

```

```
// ListeChainee.cpp
// Exemple: Liste chainée
#include <iostream.h>
#include "ListeChainee.h"

template <class T>
void Liste<T>::Ajoute(T data)
{
 Noeud<T>* nouveau = new Noeud<T>;
 nouveau->donnee = data;
 nouveau->Prochain = tete;
 tete = nouveau;

}

template <class T>
void Liste<T>::Affiche()
{
 Noeud<T>* courant = tete;
 while (courant !=0)
 {
  cout << endl << courant->donnee;
  courant = courant->Prochain;
 }
}

template <class T>
Liste<T>::~Liste()
{
 Noeud<T>* tmp;
 Noeud<T>* courant = tete;
 while (courant !=0)
 {
  tmp = courant->Prochain;
  delete courant;
  courant = tmp;
 }
}


// Reste à compléter
// bool Liste<T>::Enleve(T data)

int main()
{
 Liste<double>* dliste = new Liste<double>();

 dliste->Ajoute(3.1416);
 dliste->Ajoute(2 * 3.1416);
 dliste->Ajoute(3 * 3.1416);
 dliste->Ajoute(4 * 3.1416);

 dliste->Affiche();

 cout << endl;

 delete dliste;

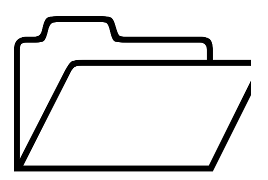
 return (0);
}
```

Le résultat affiché est :

```
12.5664
9.4248
6.2832
3.1416
```

2.1 ESPACE DES NOMS (NAMESPACES)

Bien qu'il puisse sembler toute à fait nouveau, le concept de l'**espace des noms** ont toujours existé d'une manière interne dans tous les compilateurs modernes. En effet, pour le compilateur, chaque module (fichier source) possède son propre espace de travail. Cet espace de travail comprend, entre autres, les noms des fonctions et des variables. C'est pour raison qu'il est possible de compiler séparément les modules sans problème (fichiers objets `.o` ou `.obj`). Cependant, lors de la production du fichier exécutable, l'éditeur de liens (*linker*) doit unir les fichiers objets afin de générer l'exécutable. C'est à ce moment que les différents espaces de travail sont fusionnés en un seul par l'éditeur de liens. Pour cette raison la plupart des langages n'admettent pas le partage d'un même nom par deux fonctions différentes.



Le langage C++, quant à lui, accepte la surcharge des fonctions (voir la sous-section 1.11.3 à la page 14). Cela signifie qu'il existe un dispositif interne au compilateur capable d'empêcher la fusion directe des noms par l'éditeur de liens. L'espace des noms est donc un mécanisme généralisé permettant le détournement de la fusion des noms. L'instruction qui active ce mécanisme du compilateur est l'identificateur `namespace`.

L'apport de l'instruction namespace est plus qu'appréciable si l'on considère que son utilisation permet de :

- Éviter les conflits de nom dans un programme.
- Faciliter le développement modulaire puisque le nom des fonctions et variables pourra être choisit sans susciter de conflits.
- Faciliter le développement des bibliothèques. Il n'est plus nécessaire de donner des noms particuliers aux fonctions à l'intérieur des bibliothèques. Il suffit de créer un espace de noms pour chacune des bibliothèques.

L'exemple ci-dessous démontre l'utilisation de l'instruction `namespace` :

```
#include <iostream.h>

namespace EA {
 char cVar = 'A';
}

namespace EB {
 char cVar = 'B';
}

void foo()
{
 using namespace EA;
 cout << "Dans foo(): " << EA::cVar <<
  " " << EB::cVar << " " << cVar << endl;

}

void foobar()
{
 using namespace EB;
```

```
cout << "Dans foobar(): " << EA::cVar <<
 " " << EB::cVar << " " << cVar << endl;
}
int main()
{
cout << "Dans main(): " << endl;
foo();
foobar();

using EA::cVar;
cout << cVar << endl;

// using EB::cVar;   // erreur ! cVar déjà défini
// cout << cVar << endl;

return (0);

}
```

Le résultat affiché est :

Pourquoi ce résultat ? ☞

```
Dans foo(): A B A
Dans foobar(): A B B
Dans main():
A
```

2.2 BIBLIOTHÈQUE STL

La bibliothèque STL (*Standard Template Library*) est une réalisation concrète d'une idée chère aux informaticiens : mettre, à la disposition des programmeurs, toutes les structures de données conventionnelles et avancées disponibles. À première vue, cette idée semble plutôt anodine. Or, presque tous les programmes utilisent d'une manière plus ou moins directe les structures de données. Sans l'apport d'une bibliothèque, les programmeurs sont obligés de créer eux-mêmes ces structures de données. Le temps consacré à les définir et à les construire peut représenter une grande partie du temps total d'un projet de développement.

Certes, il est possible de réunir dans un ensemble de bibliothèques toutes les implantations de toutes les structures données connues[1]. Mais ces bibliothèques auront une taille énorme. En effet, nous devons créer une copie des structures de données pour chacun des types de données. Ainsi, nous devons avoir en main une liste chaînée pour entreposer des `int`, une liste pour des `float`, une liste pour des `double`, etc. sans compter les types créés par les utilisateurs (*user-defined type*).

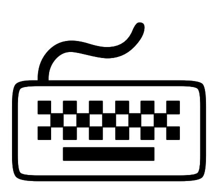
Il existe d'autres problèmes qui ne pourront pas être résolus par des bibliothèques conventionnelles. Par exemple, l'application des traitements algorithmiques à des structures de données n'est possible que par la programmation explicite de la part de l'utilisateur. Il ne peut y avoir de traitement générique applicable à un ensemble de structures.

La bibliothèque STL offre une aide généreuse aux programmeurs en mettant à leur disposition des classes « Collections » qui réalisent la plupart des structures de

[1] À condition de disposer suffisamment de temps et de patience!

données classiques et avancées. Le STL simplifie l'utilisation de ces structures de données en adoptant systématiquement les types paramétrisés (voir sous-section Types paramétrisés à la page 71) comme type de base de ses collections. De plus, la gestion de la mémoire est réalisées dans les classes collections mêmes. Les utilisateurs n'ont pas à se soucier de l'allocation dynamique de la mémoire. Enfin, un ensemble d'**algorithmes génériques** sont disponibles dans le STL. Ces algorithmes de traitement tels le tri (*sort*), la fouille (*find*), l'extraction du minimum et maximum sont applicables à la plupart des structures de données et ce, sans programmation additionnelle de la part de l'utilisateur.

L'exemple ci-dessous présente une solution à un problème pratique en utilisant le STL. Ce problème concerne le stockage des caractères entrés au clavier. Dans la solution conventionnelle, le programmeur doit réserver une zone mémoire de taille fixe pour entreposer les caractères. Par exemple, 25 octets sont réservés pour entreposer le nom d'une personne. En pratique, ce genre de limite arbitraire est souvent source de problèmes. L'utilisation de la bibliothèque STL peut éliminer ces problèmes facilement.



```
#include <iostream.h>
#include <algorithm>                   // !!
#include <vector>                      // !!
using namespace std;                   // importer l'espace des noms de STL

// objet fonctionnel
class compare {
public:
 int operator()(int x, int y) const
 {
  return (x > y);
 }
};

int main()
{
 vector<int> iVec;                     // vecteur entier
 vector<int>::iterator       i;        // iterateur entier
 int x;

 cout << "Entrer une valeur entiere" <<
  " differente de zero (0):" << endl;

 while (cin >> x, x!=0)
  iVec.push_back(x); // mettre dans le vecteur

 // algorithme générique
 sort(iVec.begin(), iVec.end(), compare());

 for (i = iVec.begin(); i != iVec.end(); i++)
  cout << *i << " ";

 cout << endl;
 return (0);

}

```

Le résultat affiché est :

```
Entrer une valeur entiere differente de zero (0):
1
34
```

```
3
65
7
2
0
65 34 7 3 2 1
```

Le programme accepte les données entrées jusqu'à l'arrivée du nombre 0 (zéro). Les données sont ensuite triées en ordre décroissant puis le résultat affiché à la sortie standard. Les éléments importants de ce programme sont énumérés ci-dessous :

- La classe vector est une classe du type paramétrisé (*template class*).
- La taille de ce vecteur est dynamique.
- La fonction push_back() ajoute un élément à la fin du vecteur.

L'énoncé vector<int>::iterator i crée une variable i spéciale appelée itérateur. Il s'agit de l'**itérateur** associé au vecteur iVec. Un itérateur se comporte comme un pointeur mais il est en fait un objet de la classe iterator de la classe vector[2]. Les lignes

```
for (i = iVec.begin(); i != iVec.end(); i++)
  cout << *i << " ";
```

utilise l'itérateur entier i et rend le code plus lisible. On peut aussi utiliser un itérateur inverse pour parcourir le vecteur à rebours :

```
vector<int>::reverse_iterator i;
for (i = iVec.rbegin(); i != iVec.rend(); i++)
cout << *i << " ";
```

La classe compare une fois instanciée est appelé **objet fonctionnel**. **Elle n'a ni constructeur, ni destructeur**. Par contre, elle possède toujours la fonction membre operator().Dans le STL, un fonctionnel sert comme outil de support pour les algorithmes génériques.

La fonction sort() est l'algorithme générique de triage. Elle s'applique sur tous les types prédéfinis de C++ et sur les classes collections appropriées. Donc, il est possible d'appliquer les algorithmes génériques à des tableaux du C++.

La déclaration de l'en-tête de la collection vecteur est :

```
<vector>
```

au lieu de

```
<vector.h>
```

[2] La classe iterator est définie à l'intérieur de la classe vector.

L'espace de nom `std` est nécessaire pour éviter les conflits de nom entre la bibliothèque STL et les autres fonctions utilisées dans le programme.

2.2.1 Composants de STL

La bibliothèque STL fait partie de la bibliothèque canonique C++ (*Standard C++ Library*). Le STL est un ensemble de classes et fonctions paramétrisées. Voici les composants de cette bibliothèque :

Collections de séquence

- `vector`
- `list`
- `deque`

Collections associatives

- `map`
- `multimap`
- `set`
- `multiset`

Algorithmes génériques

- Algorithmes de triage, de fouille, de comparaison, applicables à la plupart des collections.

Itérateurs

Pointeurs génériques servant à parcourir les collections. Ils agissent également comme interfaces entre les collections et les algorithmes génériques. Les itérateurs sont :

- `input`
- `output`
- `forward`
- `bidirectional`
- `random access`
- `istream_iterator`
- `ostream_iterator`

Fonctionnels

Toute classe ou structure qui surcharge `operator ()` est un fonctionnel.

- `plus`
- `minus`

- times
- divides
- modulus
- negate
- equal_to
- not_equal_to
- greater
- less
- greater_equal
- less_equal
- logical_end
- logical_or
- logical_not

Adapteurs

Servent à modifier le comportement des autres composants de la bibliothèque. Il existe trois types d'adapteurs.

Adapteurs de colllections

- stack
- queue
- priority_queue

Adapteurs d'itérateur

- reverse_bidirectional_iterator
- back_insert_iterator
- front_insert_iterator
- insert_iterator

Adapteurs des fonctionnels

- not1
- not2
- bind1st
- bind2nd

Enfin, la bibliothèque STL possède 13 fichiers en-tête. Ils sont présentés dans le Tableau 6.

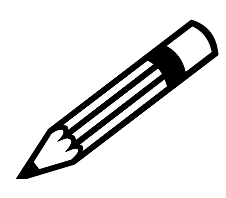
En-tête	Contenu
algorithm	Algorithmes génériques (sort, find, min, max, etc.)
deque	Structures de données semblables à une liste où l'on empile et dépile aux extrémités.
functional	Objets de support pour les algorithmes génériques.
iterator	Itérateurs des structures de données.
list	Listes chaînes (double ou simple).
map	Collections associatives.
memory	Gestion dynamique de la mémoire.
numeric	Fonctions numériques, somme cumulative, produit de sommes, etc.
queue	Structures de données sous forme de queue.
set	Structures de données réalisant des ensembles.
stack	Piles informatiques.
utility	Fonctions de support pour le STL.
vector	Structures de données sous forme de vecteur.

Tableau 6 Fichiers en-tête de la bibliothèque STL.

2.2.2 COMPATIBILITÉ ET PORTABILITÉ

La bibliothèque STL est disponible pour tous les environnement de développement C++ modernes. Les programmes C++ utilisant le STL sont donc compatibles avec tous les compilateurs commerciaux. Cependant, pour des raisons historiques, certains compilateurs n'acceptent pas l'usage de <vector> à la place il faut écrire <vector.h>. Dans de très rares cas, l'usage de using namespace std n'est pas accepté et n'est pas nécessaire.

2.2.3 RÈGLES D'UTILISATION



Pour pouvoir profiter des capacités de la bibliothèque STL dans nos programmes, il est nécessaire de créer des classes et objets disposant des caractéristiques minimales suivantes :

- Un constructeur de copie.

  Le constructeur de copie est utilisé par le STL dans le copiage des objets dans les collections. Rappel : un constructeur de copie est un constructeur qui ne contient qu'un seul paramètre soit une référence à la classe elle-même et dont le rôle est de copier le contenu des variables membres de la classe. Par exemple :

```
class foo {
public :
 foo() { a = 1; b = 2; }      // constructeur par défaut
 foo(const foo& FooObj)       // constructeur de copie
 {

  a = FooObj.a;
  b = FooObj.b;
```

```
}
public:
 int a,b;
};
```

- Un opérateur d'assignation =.

  L'opérateur d'assignation est utilisé par le STL dans l'assignation des objets à d'autres objets temporaires. Voici l'exemple d'une déclaration de cet opérateur :

```
class foo {
public :
 foo() { a = 1; b = 2; }      // constructeur par défaut
 foo(const foo& FooObj)       // constructeur de copie
 {
  a = FooObj.a;
  b = FooObj.b;
 }
foo& operator=(foo& rhs)      // opérateur d'assignation
{
a = rhs.a;
b = rhs.b;
return *this;
}

public:
 int a, b;
};
```

- Un opérateur relationnel d'égalité ==.

  L'opérateur d'égalité permet à la bibliothèque STL d'effectuer des comparaisons sur des objets. Le STL peut construire les autres opérateurs relationnels à l'aide de l'opérateur « plus petit que » et de l'opérateur de négation.

- Un opérateur relationnel de comparaison « plus petit que » <.

  L'opérateur « plus petit que » permet à la bibliothèque STL d'effectuer des comparaisons sur des objets. Le STL peut construire les autres opérateurs relationnels à l'aide de l'opérateur d'égalité et de l'opérateur de négation.

2.3 ALGORITHMES GÉNÉRIQUES

Les algorithmes génériques sont des algorithmes de traitement applicables à des classes collections du STL et des types prédéfinis de C++. Ces algorithmes sont déclarés dans le fichier `algorithm`. Voici un exemple utilisant l'algorithme générique `sort()`.

```
class compare {
public:
 int operator()(int x, int y) const

 {
  return (x > y);
```

```
}
};
int main()
{
vector<int> iVec;                    // vecteur entier
vector<int>::iterator       i;      // iterateur entier
int x;
cout << "Entrer une valeur entiere" <<
 " differente de zero (0):" << endl;
while (cin >> x, x!=0)
 iVec.push_back(x); // mettre dans le vecteur
// algorithme générique sort
sort(iVec.begin(), iVec.end(), compare());
for (i = iVec.begin(); i != iVec.end(); i++)
 cout << *i << " ";
cout << endl;
return (0);
}
```

La façon d'utiliser la fonction `sort()` montre bien qu'elle est une fonction au type paramétrisé. Le triage suppose qu'il existe des opérateurs de comparaison pour les types manipulés. En effet, tous les opérateurs relationnels sont définis pour les types de base de C++. Encore une fois, pour les types définis par le programmeur (*user-defined types*), nous devons respecter les règles suivantes :

- Surcharger les opérateurs == et < seulement.
- Utiliser les objets fonctionnels.

D'abord, pourquoi seulement deux opérateurs surchargés? La raison est que le STL est en mesure de générer les autres relations à partir de `==` et `<`. En appliquant l'algèbre élémentaire, nous obtenons les identités suivantes :

```
x != y → !(x == y)
x > y  → y < x
x <= y → !(y < x)
x >= y → !(x < y)
```

Enfin, le code ci-dessous est un autre exemple utilisant un algorithme générique. Cette fois, l'algorithme `find()` est utilisé pour trouver une valeur quelconque dans un vecteur d'entiers.

```
// Exemple d'un algorithme générique
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
```

```
vector<int> iVec;                   // vecteur entier
vector<int>::iterator       i;      // iterateur entier
int x;

cout << "Entrer une valeur entiere" <<
 " differente de zero (0):" << endl;

while (cin >> x, x!=0)
 iVec.push_back(x); // mettre dans le vecteur

cout << "Valeur à trouver: "; cin >> x;

// algorithme générique pour la fouille
i = find(iVec.begin(), iVec.end(), x);

if (i ==iVec.end())
 cout << "n'est pas dans le vecteur" << endl;
else {
 cout << "trouve";
 if (i == iVec.begin())
  cout << " comme le premier element";
 else
  cout << " apres " << *--i;
}
cout << endl;

return (0);

}
```

Le résultat affiché est :

```
Entrer une valeur entiere differente de zero (0):
1
2
3
4
5
6
0
Valeur a trouver: 5
trouve apres 4
```

Enfin, à titre d'exemple, le tableau ci-dessous donne les algorithmes génériques les plus utilisés pour la classe collection `vector`.

Algorithme	Description
`fill()`	Assigner au vector une valeur initiale.
`copy()`	Copier une séquence d'éléments dans une autre séquence.
`max_element()`	Trouver le plus grand élément d'une collection.
`min_element()`	Trouver le plus petit élément d'une collection.
`reverse()`	Inverser l'ordre des éléments dans une séquence.
`count`	Compter le nombre d'éléments égal à une valeur dans une séquence.
`count_if`	Compter le nombre d'éléments dont la comparaison est représentée par un fonctionnel..
`transform()`	Transformer (modifier) les éléments d'une séquence à l'aide d'un fonctionnel passé en paramètre.
`find()`	Trouver un élément et retourner un itérateur représentant la position de l'élément dans la séquence.

`find_if()`	Trouver un élément dont la comparaison est représentée par un fonctionnel et retourner un itérateur représentant la position de l'élément dans la séquence.
`replace()`	Remplacer un élément de la séquence par un autre élément .
`replace_if()`	Remplacer un élément de la séquence, dont la comparaison est représentée par un fonctionnel, par un autre élément.
`sort()`	Trier la collection.
`for_each()`	Exécuter le fonctionnel passé en paramètre sur chacun des éléments de la collection.
`iter_swap()`	Échanger les éléments pointés par deux itérateurs.

Tableau 7 Algorithmes génériques pour la classe `vector`.

2.4 CATÉGORIES D'ITÉRATEURS

Les itérateurs sont des objets servant à parcourir les différentes classes collections du STL. Leur comportement ressemble beaucoup à une variable que l'on utilise pour indexer un tableau. Cependant, comme nous allons voir, les itérateurs sont beaucoup plus puissants que les simples indices. On définit les itérateurs selon les d'opérations disponibles. Toutes les catégories d'itérateurs possèdent trois opérations de base. C'est-à-dire, égalité, non égalité et assignation. Dans l'encadré ci-dessous, `i`, `j` sont des objets itérateurs.

```
i == j
i != j
i = j
```

Les différentes catégories d'itérateurs sont regroupées par les opérations supplémentaires. Ces catégories sont montrées dans le Tableau 8. Ce tableau contient trois (3) colonnes. La colonne désignée « Opérations supplémentaires » indique la liste des opérations que l'on peut effectuer en plus des trois opérations de base. L'algorithme générique écrit entre parenthèses est cité comme un exemple qui utilise cet catégorie d'itérateurs. La colonne « Classes collections applicables » indique lesquelles des collections sont utilisables avec la catégorie d'itérateurs donnés. Toujours dans ce tableau `i`, `j` sont des objets itérateurs et `x` est une variable de même type que la classe collection et `n` est un entier

		Classes collections applicables			
Catégorie	**Opérations supplémentaires**	**Liste**	**Vecteur**	**Deque**	**Tableau**
Itérateur d'entrée	`x = *i`, `++i`, `i++` (`sort`)	√	√	√	√
Itérateur de sortie	`*i = x`, `++i`, `i++` (`copy`)	√	√	√	√
Itérateur vers l'avant	Identique à entrée et à sortie (`replace`)	√	√	√	√
Itérateur bidirectionnel	Identique à vers l'avant plus `--i` et `i++` (`reverse`)	√	√	√	√
Itérateur à accès aléatoire	Identique à bidirectionnel avec `i + n`, `i - n`, `i += n`, `i -= n`, `i < j`, `i > j`, `i <= j`, `i >= j` (`sort`)	non	√	√	√

Tableau 8 Catégorie d'itérateurs dans le STL.

2.4.1 ITÉRATEURS ET ALGORITHMES GÉNÉRIQUES

Le Tableau 9 explicite quelques algorithmes génériques et les itérateurs nécessaires.

Algorithme générique	Itérateurs utilisés
`sort`	Itérateurs d'entrée, itérateurs à accès aléatoire
`copy`	Itérateur de sortie
`replace`	Itérateur vers l'avant
`reverse`	Itérateur bidirectionnel

Tableau 9 Quelques algorithmes génériques et ses itérateurs.

2.4.2 ITÉRATEURS ET LES COLLECTIONS

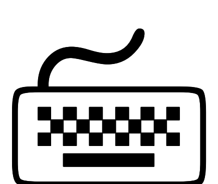
Les itérateurs sont fortement associés à des classes collections. Après tout les itérateurs servent à parcourir les collections. Voici un tableau résumant les collections et les itérateurs utilisables.

Itérateurs	Collections applicables
D'entrée	`list`, `vector`, `deque` et tableau.
De sortie	`list`, `vector`, `deque` et tableau.
Vers l'avant	`list`, `vector`, `deque` et tableau.
Bidirectionnel	`list`, `vector`, `deque` et tableau.
À accès aléatoire	`vector`, `deque` et tableau.

Tableau 10 Catégories d'itérateurs et quelques collections applicables.

2.5 ITÉRATEURS DES FLUX

Il existe un itérateur spécial pour traiter les entrées/sorties. Les **itérateurs de flux** sont fort utiles pour les algorithmes génériques. À l'aide de ces itérateurs, l'affichage à l'écran et l'enregistrement des données sur disque peut être réalisé succinctement et d'une manière élégante. Voici un exemple d'utilisation d'un itérateur de flux.



```
// Exemple d'un itérateur de flux
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
 const int N = 4;
 int a[N] = {7, 6, 9, 2};
 copy(a, a+N, ostream_iterator<int>(cout, " "));
 cout << endl;

 return (0);

}
```

Le résultat affiché est :

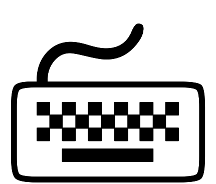
```
7 6 9 2
```

La ligne 10 copie une séquence de données source vers une destination quelconque.

```
copy(a, a+N, ostream_iterator<int>(cout, " "));
```

Dans cet exemple, `a` représente le début du vecteur et `a+N` représente la fin du vecteur à copier. L'itérateur de flux est `ostream_iterator<int>(cout, " ")` où `cout` est le flux associé à l'itérateur et `" "` (caractère espace) est le séparateur utilisé dans l'affichage. Enfin, l'itérateur de flux est réalisé par une classe paramétrisée.

Le deuxième exemple ci-dessous utilise un itérateur de flux pour afficher le résultat des données triées par `sort`. L'itérateur de flux est associé à `cout` (sortie standard) et l'algorithme générique `copy` est utilisé pour déplacer les données triées.



```
// Exemple d'un itérateur de flux (2)
#include <iostream>
#include <algorithm>
using namespace std;

bool FuncCompare(const int x, const int y)
{
 return (x > y);
}


int main()
{
 const int N = 8;
 int a[N] = {1234, 5432, 8943, 3346, 9831, 7842, 8863, 9820};

 cout << "Vecteur avant triage:\n";
 copy(a, a+N, ostream_iterator<int>(cout, " "));
 cout << endl;

 // Triage
 sort(a, a+N, FuncCompare);

 cout << "Vecteur après triage:\n";
 copy(a, a+N, ostream_iterator<int>(cout, " "));
 cout << endl;

 return (0);

}
```

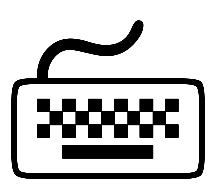
Le résultat affiché est :

```
Vecteur avant triage:
1234 5432 8943 3346 9831 7842 8863 9820
Vecteur apres triage:
9831 9820 8943 8863 7842 5432 3346 1234
```

2.6 OBJETS FONCTIONNELS

Les objets fonctionnels ont déjà été appliqués dans de nombreux exemples de ce chapitre. Un fonctionnel est une classe dans laquelle l'opérateur d'appel `operator()` est définie. Il ne possède pas de constructeur ni de destructeur explicite pour les

objets fonctionnels. Donc, un fonctionnel **n'est pas** une entité réservée à la bibliothèque STL; il est possible de les définir dans tout programme C++. De plus, un fonctionnel est beaucoup plus général qu'une surcharge d'opérateur puisqu'il est un objet. Par conséquent, il est possible de les utiliser comme paramètre à un autre objet ou fonction. Ci-dessous est un exemple utilisant un objet fonctionnel.



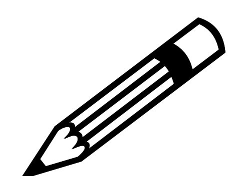
```
// Exemple d'utilisation de fonctionnel
#include <iostream.h>

class Compare {
public:
 int operator()(int x, int y) const
 {
  return (x > y);
 }
};

int main()
{
 Compare V;
 cout << V(2, 15) << endl;
 cout << Compare()(5,3) << endl;
 return (0);
}
```

Le résultat affiché est :

```
0
1
```



Observer attentivement l'exemple ci-dessus. Puisque `operator()` est défini avec deux (2) entiers pour la classe `Compare`, on peut écrire :

```
V(2, 15);
```

où `V` est un objet de la classe `Compare`. Il s'agit d'une abréviation pour :

```
V.operator(2, 15);
```

On peut aussi écrire

```
Compare()(2, 15)
```

puisque `Compare()` invoque le constructeur par défaut de la classe `Compare`. Comme `operator()(int, int)` existe dans la classe, `Compare()(2, 15)` est donc une construction légale.

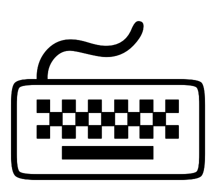
Une question fort pratique ☞

Alors comment peut-on différencier ces écritures des fonctionnels ? Pour répondre à cette question, définissons la classe suivante comme modèle.

```
class Carre {
public:
  int operator()(int x) const
  {
  return x * x;
  }
};
```

- `Carre` est une classe (un type);
- `Carre()` est un objet fonctionnel (un fonctionnel);
- `Carre()(8)` est un appel de fonction.

Un fonctionnel peut aussi servir comme argument à une classe paramétrisée (*template*). Voici un exemple de cette utilisation. Le fonctionnel `Carre()` retourne une valeur au carré de `x` tandis que le fonctionnel `Cube()` retourne la valeur au cube de `x`. La classe paramétrisée `Foo` possède un constructeur et une fonction membre publique `Affiche()`. La classe `Foo` contient également une variable privée `iVar`. Cette variable privée de `Foo` est initialisée lors de l'appel du constructeur.



```
// Exemple d'utilisation de fonctionnel (2)
#include <iostream.h>

class Carre {
public:
 int operator()(int x) const
 {
  return (x * x);
 }
};

class Cube {
public:
 int operator()(int x) const
 {
  return (x * x * x);
 }
};

template <class T>
class Foo {
public:
 Foo(int i)
  : iVar(i)
 { }
 void Affiche() const
 {
  cout << T()(iVar) << endl;
 }
private:
 int iVar;
};

int main()
{
 Foo<Carre> xCarre(10);
 xCarre.Affiche();

 Foo<Cube> xCube(10);
 xCube.Affiche();

 return (0);

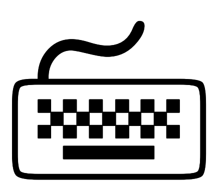
}
```

Le résultat affiché est :

```
100
1000
```

La classe `Foo` sert à entreposer une donnée (`iVar`). Le type paramétrisé de la classe `Foo` est un fonctionnel `Carre` ou `Cube` (ligne 36 et 39). Les fonctionnels `Carre` et `Cube` servent à appliquer des transformations sur la donnée de `Foo`.

Il est possible de donner plus d'un paramètre à des fonctionnels. Cependant, à cause de son utilisation particulière dans le STL, il est rare d'avoir des fonctionnels avec plus que deux paramètres. L'exemple ci-dessous met en œuvre des fonctionnels à deux paramètres.



```
// Exemple d'utilisation de fonctionnel (3)
#include <iostream.h>

template <class T>
class MoinsQue {
public:
 bool operator() (const T& x, const T& y) const
 {
  return x < y;
 }
};

class CompareDernierChiffre {
public:
 bool operator() (int x, int y) const
 {
  return (x % 10 < y % 10);
 }
};

template <class T, class Compare>
class ChoisirPair {
public:
 ChoisirPair(const T& x, const T& y)
  : a(x), b(y)
 { }
 void AfficheLePlusPetit() const
 {
  cout << (Compare()(a, b) ? a : b) << endl;
 }
private:
 T a, b;
};

int main()
{
 ChoisirPair<double, MoinsQue<double> > foo1(4321.4,
↳ 98.7);
 foo1.AfficheLePlusPetit();


 ChoisirPair<int, CompareDernierChiffre> foo2(431, 77);
 foo2.AfficheLePlusPetit();

 return (0);

}
```

Le résultat affiché est :

```
98.7
431
```

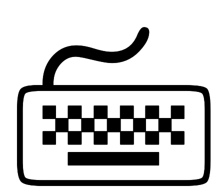
Fonctionnels uniaires et adaptateurs ☞

Dans le STL, il existe également des **fonctionnels unaires**. Ces fonctionnels unaires premetttent la réalisation des relations de type α R c où α est une variable, R une relation et c une valeur constante. Par exemple $x < 100$ est une telle relation.

Un fonctionnel unaire de type `x < 100` est un fait un cas particulier de `x < y` avec `y = 100`. Pour réaliser l'assignation de la partie constante `y = 100`, on utilise un **adaptateur** du STL. Ainsi,

```
bind2nd(less<int>(), 100)
```

est un adaptateur qui réalise `x < 100`. `bind2nd` (lire *bind second member*) est un adaptateur qui lie le 2e paramètre du fonctionnel `less` à la valeur 100. Le fonctionnel `less` dans ce cas est du type `int` et comme son nom l'indique, il réalise la relation « plus petit que ».



```
1. // Exemple d'utilisation d'adaptateur
2. #include <iostream.h>
3. #include <algorithm>
4. #include <functional>
5.
6. using namespace std;
7.
8. int main()
9. {
10. int iVec[] = {750, 2, 11, 5, 100, 19, 6, 7, 103, 1};
11. int Resultat;
12.
13. Resultat = count_if(iVec, iVec+10, bind2nd(less<int>(), 100));
14. cout << "Nombre de chiffre plus petit que 100: " <<
15.  Resultat << endl;
16.
17. return (0);
18.
19. }
```

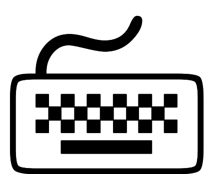
Le résultat affiché est :

```
Nombre de chiffre plus petit que 100: 7
```

Dans cet exemple, l'algorithme générique `count_if` (à la ligne 13) parcoure le tableau `iVec` et applique à chacun de ses éléments le fonctionnel unaire `bind2nd(less<int>(), 100)`. Le nombre d'éléments plus petits que 100 est comptabilisé puis affiché à la sortie standard.

Évidemment, il existe également un adaptateur pour réaliser l'expression `y < x` où `y` est une constante. Il s'agit de `bind1st` (lire *bind 1st member*). Quant à la négation de forme `!(x < y)`, on peut utiliser un **adaptateur négateur** :

```
not2(less<int>())
```



```
// Exemple d'utilisation d'adaptateur (2)
#include <iostream.h>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
 int iVec[] = {750, 2, 11, 5, 100, 19, 6, 7, 103, 1}, i;

 sort(iVec, iVec+10, not2(less<int>()));

 for (i = 0; i < 5; i++) cout << iVec[i] << " ";
 cout << endl;

 return (0);

}
```

Le résultat affiché est :

```
750 103 100 19 11 7 6 5 2 1
```

Si l'on modifiait le fonctionnel `less` par `greater`, on aurait le résultat suivant :

```
1 2 5 6 7 11 19 100 103 750
```

Évidemment, on peut aussi appliquer un négateur à un fonctionnel unaire :

```
not1(bind2nd(less<int>(), 100))
```

Enfin, les fonctionnels `less` et `greater` sont des fonctionnels prédéfinis de STL. L'exemple ci-dessous montre l'application d'un adaptateur à des fonctionnels définis par le programmeur.

```
// Exemple d'utilisation d'adaptateur (3)
#include <iostream.h>
#include <algorithm>
#include <functional>

using namespace std;

struct MoinsQue100 : unary_function<int, bool> {
 bool operator()(int x) const
 {
  return x < 100;
 }
};

int main()
{
 int iVec[] = {750, 2, 11, 5, 100, 19, 6, 7, 103, 1};

 int Resultat = count_if(iVec, iVec+10, not1(MoinsQue100()));
cout << " Nombre de chiffre plus grand que 100 : "  << Resultat << endl;

 return (0);

}
```

Le résultat affiché est :

```
Nombre de chiffre plus grand que 100 : 3
```

Création des fonctionnels acceptant des adaptateurs et la liste des fonctionnels prédéfinis

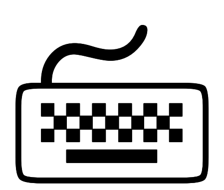


Pour créer un fonctionnel acceptant des adaptateurs, nous devons dériver à partir de la classe paramétrisée `unary_function` ou `binary_function` de STL. Nous avons vu que `less` et `greater` sont des fonctionnels prédéfinis de STL. Voici la liste complète de ces fonctionnels prédéfinis :

fonctionnels	**Signification**
plus<T>	Réalise l'opérateur +.(addition).
minus<T>	Réalise l'opérateur -(soustraction).
modulus<T>	Réalise l'opérateur modulo.
times<T>	Réalise l'opérateur *.
divides<T>	Réalise l'opérateur /.
equal_to<T>	Réalise l'opérateur ==.
not_equal_to<T>	Réalise l'opérateur !=.
greater<T>	Réalise l'opérateur >.
less<T>	Réalise l'opérateur <.
greater_equal<T>	Réalise l'opérateur ≥.
less_equal<T>	Réalise l'opérateur ≤.
logical_and<T>	Réalise l'opérateur ET-logique (&&)
logical_or<T>	Réalise l'opérateur OU-logique (||)
negate<T>	Réalise l'opérateur complément (a → -a)
logical_not<T>	Réalise l'opérateur NON-logique (!).

Tableau 11 Liste des adaptateurs prédéfinis dans STL.

Ces fonctionnels sont de nature arithmétique (et logique). Ainsi, il est possible d'effectuer très simplement certaines opérations arithmétiques :



```
// Exemple d'utilisation de fonctionnel arithmérique
#include <iostream.h>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
 float fVec[] = {10.1f, 22.3f, 41.4f, -37.4f, \

  76.2f, 31.8f, -8.3f, -19.1f};
 float fVec2[8];

transform(fVec, fVec+8, fVec2, negate<float>());
 int i;
 for(i = 0; i < 8; i++) cout << fVec2[i] << " ";
 cout << endl;

 return (0);

}
```

Le résultat affiché est :

```
-10.1 -22.3 -41.4 37.4 -76.2 -31.8 8.3 19.1
```

Dans l'exemple ci-dessus, un tableau de `float` est complémenté par le fonctionnel `negate` en utilisant l'algorithme générique `transform`. Voici un autre exemple (Pourquoi pas!) :

```
// Exemple d'utilisation de fonctionnel arithmérique
#include <iostream.h>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
 float fVec[] = {10.1f, 22.3f, 41.4f, -37.4f, \
  76.2f, 31.8f, -8.3f, -19.1f};

float fVec2[] = { 2.0f, 2.1f, 2.2f, 4.3f, 5.5f, 6.2f, \
  7.8f, 8.2f};
 float fResultat[8];

 transform(fVec, fVec+8, fVec2, fResultat,
 divides<float>());
 int i;
 for(i = 0; i < 8; i++) cout << fResultat[i] << " ";
 cout << endl;

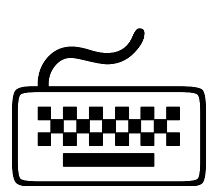
 return (0);

}
```

Le résultat affiché est :

```
5.05 10.619 18.8182 -8.69767 13.8545 5.12903 -1.0641 -2.32927
```

L'algorithme générique `transform` permet la mise en œuvre de certains fonctionnels. L'utilisation judicieuse des fonctionnels élimine les nombreuses boucles nécessaires dans le calcul numérique. Il est également possible d'utiliser l'algorithme générique transform avec les fonctionnels définis par le programmeur.



```
// Exemple d'utilisation de fonctionnel arithmérique (2)
#include <iostream.h>
#include <algorithm>
#include <functional>

using namespace std;

struct Calcul : binary_function<float, float ,float>
{
 float operator()(float x, float y) const
 {
  return 2.0f * x + y; // 2x + y
 }
};


int main()
{
```

```
float fVec[] = {10.1f, 22.3f, 41.4f, -37.4f, \
 76.2f, 31.8f, -8.3f, -19.1f};
float fVec2[] = { 2.0f, 2.1f, 2.2f, 4.3f, 5.5f, 6.2f, \
 7.8f, 8.2f};
float fResultat[8];

transform(fVec, fVec+8, fVec2, fResultat, Calcul());
int i;
for(i = 0; i < 8; i++) cout << fResultat[i] << " ";
cout << endl;

return (0);

}
```

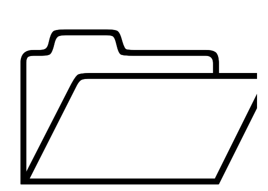
Le résultat affiché est :

```
22.2 46.7 85 -70.5 157.9 69.8 -8.8 -30
```

2.7 COLLECTIONS DE SÉQUENCE

Les listes (`list`), vecteurs (`vector`), deques (`deque`) et tableaux sont des **collections de séquence**. Associer à ces collections est un type `value_type`. Le type `value_type` représente le type des objets entreposés dans une collection. Puisque `value_type` est défini à l'intérieur d'une collection, nous devons écrire :

```
vector<double>::value_type
```



L'utilité de `value_type` est de permettre la création d'un objet de même type que ceux entreposés dans la collection sans connaître la nature exacte de leur type. Ainsi, pour créer un objet de même type que, ceux contenus dans une collection, on peut écrire simplement :

```
vector<double>::value_type UnObjetDansLeVector;
```

Les types prédéfinis à l'intérieur des collections de séquence sont donnés dans le tableau suivant.

type prédéfini	Signification
`value_type`	Le même type que les objets entreposés dans la collection.
`reference`	Une référence de même type que les objets entreposés dans la collection.
`const_reference`	Une référence constante de même type que les objets entreposés dans la collection.
`iterator`	Un itérateur de même type que les objets entreposés dans la collection.
`const_iterator`	Un itérateur constant de même type que les objets entreposés dans la collection.
`reverse_iterator`	Un itérateur de même type que les objets entreposés dans la collection mais permet le parcours à rebours.

const_reverse_iterator	Un itérateur constant de même type que les objets entreposés dans la collection mais permet le parcours à rebours.
difference_type	Le type représentant la différence de deux objets utilisés dans la comparaison des grandeurs.
size_type	Un type pour entreposer une grandeur.
vector_allocator	Le type de l'objet qui effectue l'allocation dynamique de la mémoire dans les collections de séquence.

Tableau 12 Liste des adaptateurs prédéfinis dans STL.

2.7.1 VECTEURS

Dans le STL, un vecteur est une collection de séquences dans laquelle l'insertion et l'élimination des éléments sont plus efficaces si elles sont réalisées à la fin de l'espace d'entreposage. D'abord, voici comment instancier des objects de la classe `vector` :

```
// vecteur dynamique d'entiers avec valeur initiale 0
vector<int> iVec(5, 0);        // taille initiale 5
// Vecteur dynamique de simple précision sans valeur initiale
vector<float> fVec(20);        // taille initiale 20
// Vecteur dynamique contenant des objets de type foo sans valeur initiale
vector<foo> FooVec(10); // taille initiale 10
// Vecteur dynamique contenant des pointeurs de type foo avec valeur initiale 0
vector<foo *> FooPtrVec(15, NULL);     // taille initiale 15
```

Il existe deux opérateurs d'indexage dans une collection `vector` :

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Par exemple, si le vecteur est du type `int` alors le type `reference` est simplement `int&`. De plus, la mémoire utilisée est gérée automatiquement par la classe `vector`. Pour réaliser efficacement l'allocation dynamique de la mémoire, `vector` pré-alloue un bloc de mémoire pour permettre l'insertion rapide des éléments. La figure montre l'organisation d'un vecteur `v` et ses divers paramètres sous forme de fonctions membres.

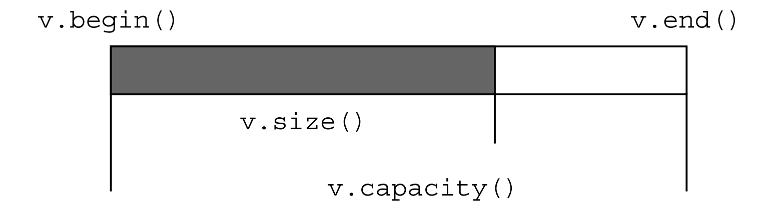


Figure 7 Taille et allocation de la mémoire des vecteurs dans STL.

On peut obtenir la quantité totale de la préallocation par la fonction membre `capacity()`. Pour connaître le nombre d'éléments dans le vecteur, on peut utiliser la fonction membre `size()`. L'exemple ci-dessous permet de mieux comprendre le schème de préallocation des vecteurs.

```
// Exemple d'allocation interne de la mémoire des vecteurs
```

```
#include <iostream.h>
#include <iomanip.h>
#include <vector>

using namespace std;

int main()
{
 vector<int> iVec;
 vector<int>::size_type indice1 = 12345, indice2;
 long i;

 cout << "iVec.size()\tiVec.capacity()\n";
 for (i = 0L; i < 1000L; i++)
 {
  indice2 = iVec.capacity();
  if (indice2 != indice1)
  {
   cout << setw(8) << iVec.size() << "        "
      << setw(8) << indice2 << endl;
   indice1 = indice2;
  }
  iVec.push_back(234);
 }

 return (0);

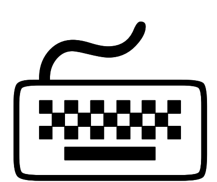
}
```

Le résultat affiché est :

Que peut-on conclure en examinant ces résultats? ☞

```
iVec.size()     iVec.capacity()
       0              0
       1              1
       2              2
       3              4
       5              8
       9             16
      17             32
      33             64
      65            128
     129            256
     257            512
     513           1024
```

Enfin, pour donner une grandeur initiale aux vecteurs, utiliser la fonction `reserve()`. L'indexage dans un vecteur STL est semblable à l'indexage d'un tableau en C++.



```
// Exemple STL utilisant la classe vector (avec indexage [])
#include <iostream.h>
#include <algorithm>
#include <vector>

using namespace std;

// objet fonctionnel
class compare {
public:
 int operator()(int x, int y) const
 {
  return (x > y);
 }
};

int main()
```

```
{
vector<int> iVec;
int x;

cout << "Entrer une valeur entiere" <<
 " differente de zero (0):" << endl;

while (cin >> x, x!=0)
 iVec.push_back(x);// mettre dans le vecteur

// algorithme générique
sort(iVec.begin(), iVec.end(), compare());

x = 0; // redondant
while (x < iVec.size()) {
 cout << iVec[x] << " ";
 x++;
}

cout << endl;
return (0);

}
```

Le résultat affiché est :

```
Entrer une valeur entiere differente de zero (0):
123
54
656
67
389
321
0
656 389 321 123 67 54
```

À noter que l'utilisation des opérateurs `[]` facilite la lecture du code source surtout pour les novices.

La bibliothèque STL nous propose l'utilisation des itérateurs pour parcourir une collection telle le vecteur. Voici un exemple montrant le parcours d'un vecteur d'entiers nommé `iVec` par opposition à celui d'un tableau d'entiers C++ nommé `iTab` :

```
1. // Parcours d'un tableau C++ iTab
2. for (int nIter = 0; nIter < MAX_ELEMENT; nIter++)
3.  cout << iTab[nIter] << endl;
4.
5. // Parcours d'un vecteur STL iVec
6. vector<int>::iterator Iter;  // obtenir un itérateur du vecteur d'entiers
7. for (Iter == iVec.begin(); Iter != iVec.end(); Iter++)
8.  cout << *Iter << endl;
```

À la ligne 5 il y a création d'un objet itérateur de même type que le vecteur `iVec` (on sous-entend que `iVec` est de type `vector<int>`). Le parcours du vecteur `iVec` est réalisé à l'aide de la boucle for à la ligne 7. Attention ! Les opérateurs relationnels autres que `!=` ne sont pas surchargés dans les itérateurs. Donc, nous devons toujours comparer les limites de bouclage à l'aide de l'opérateur d'inégalité.

Voici la liste des fonctions membres de la classe STL vector. Rappelons-nous que **l'utilisation d'un vecteur est la plus efficace lorsque l'on insère et enlève les éléments à partir de la fin du vecteur**. Observer bien les services offerts par cette classe fort utile. On peut apprendre beaucoup sur la conception orientée objet en examinant simplement les techniques utilisées dans la bibliothèque STL.

2.7.2 FONCTIONS MEMBRES DES VECTEURS

Constructeurs et destructeur

Vector()	Vector<int> iVec;
Vector(size_type n, const T& value = T());	Vector<int> iVec(5, -3) crée un vecteur de 5 éléments de valeur –3. vector<int> iVec(5) crée un vecteur de 5 éléments.
Vector(const vector<T>& x);	Vector<int> V(iVec) crée un vecteur à partir d'un autre vecteur.
Vector(const_iterator first, const_iterator last);	Vector<int> V(iVec.begin()+2, iVec.begin()+4) crée un vecteur à partir d'un sous-ensemble des éléments d'un autre vecteur.
~vector();	Destructeur de la classe.
Opérateurs	
Reference operator[](size_type n);	Accès aléatoire. vector<int> iVec(5); int x = iVec[2];
const_reference operator[](size_type n) const;	Version const.
Services	
iterator begin()	Itérateur qui retourne le premier élément du vecteur.
iterator end()	Itérateur qui retourne le dernier élément du vecteur.
void push_back(const T& x);	Ajoute un élément à la fin du vecteur.
reverse_iterator rbegin()	Iterateur pour le parcours à rebours.
reverse_iterator rend()	Iterateur pour le parcours à rebours.
const_iterator begin() const;	version const.
const_iterator end() const;	version const.
const_reverse_iterator rbegin() const;	version const.
const_reverse_iterator rend() const;	version const.
size_type size()const;	Retourne la grandeur logique du vecteur.

size_type capacity() const;	Retourne la grandeur physique du vecteur.
void reserve(size_type n);	Règle la grandeur physique du vecteur.
size_type max_size() const;	Retourne la grandeur physique maximale du vecteur.
bool empty() const;	Indique si le vecteur est vide ou non.
reference front();	Retourne une référence du premier élément du vecteur.
reference back();	Retourne une référence du dernier élément du vecteur.
const_reference front();	Version const.
const_reference back();	Version const.
void swap(vector<T>& x);	Échange les éléments de deux vecteurs.
iterator insert(iterator position, const T& x);	Insère un élément `x` dans le vecteur à la position indiquée par `position`.
void insert(iterator position, const_iterator first, const_iterator last);	Insère un ensemble d'éléments de `first` à `last` dans le vecteur à partir de la position indiquée par `position`.
void insert(iterator position, size_type n, const T& x);	Insère `n` fois élément `x` dans le vecteur à partir de la position indiquée par `position`.
void pop_back();	Enlève le dernier élément de la liste.
void erase(iterator position);	Enlève l'élément à la position indiquée par `position`.
void erase(iterator first, iterator last);	Enlève les éléments dans l'intervalle `first` et `last`.

2.7.3 DEQUE

Un deque est une collection semblable à un vecteur. **Cependant, on peut ajouter et enlever efficacement les éléments à la tête et à la fin d'un deque**. Il possède les mêmes fonctions membres que le vecteur cependant les fonctions ci-dessous sont propres à un deque.

2.7.4 FONCTIONS PROPRES AUX DEQUES

Void push_front(const T& x);	Place l'élément `x` dans la tête du deque.
Void pop_front();	Enlève l'élément à la tête du deque.

2.7.5 LISTES

La classe `list` est semblable à la classe `vector` et `deque`. **Cette classe permet l'insertion et élimination d'un élément en un temps constant peu importe sa position**. Cependant, l'accès aléatoire **n'est pas possible** à l'intérieur d'une liste. De plus, l'algorithme générique `sort` ne s'applique pas à une liste.

2.7.6 FONCTIONS PROPRES AUX LISTES

void sort();	Fonction interne à une liste pour le triage.
void unique();	Enlève les duplications de la liste.
void splice(iterator position, list<T>& x);	`L.splice(i, M);` déplace les éléments de la liste `M` dans la liste `L` en avant de la position `i`.
void splice(iterator position, list<T>& x, iterator j);	`L.splice(i, M, j);` déplace l'élément à la position `j` de la liste `M` dans la liste `L` à la position `i`.
void splice(iterator position, list<T>& x, iterator first, iterator last);	`L.splice(i, M, j1, j2);` déplace les éléments entre `j1` et `j2` de la liste `M` dans la liste `L` à la position avant `i`.
void remove(const T& value);	Enlève tous les éléments `value` de la liste.
void reverse();	Inverse l'ordre des éléments de la liste.
void merge(list<T>& x);	Joindre deux listes en une seule.

2.8 COLLECTIONS ASSOCIATIVES

Les classes `set`, `multiset`, `map` et `multimap` forment l'ensemble des **collections associatives** de STL. Une collection associative est une collection d'objets qui ne sont pas nécessairement ordonnés[3]. L'avantage des collections associatives est la possibilité d'associer un élément de la collection à une clé. L'accès d'un élément via une clé est donc beaucoup plus rapide. De plus, les clés utilisées n'ont pas à être numériques d'où une plus grande flexibilité que les collections de séquence. Leurs déclarations sont montrées ci-dessous :

Déclarations des classes réalisant différentes collections associatives ☞

```
template <class Key, class Compare>
class set {
  : :
  : :
};


template <class Key, class Compare>
class multiset {
  : :
  : :
```

[3] Par contre, les éléments sont toujours triés d'une manière interne, par ces collections.

```
};


template <class Key, class T, class Compare>
class map {
  : :
  : :
};


template <class Key, class T, class Compare>
class multimap {
  : :
  : :
};
```

2.8.1 SET ET MULTISET

Les collections `set` et `multiset` possèdent deux types paramétrisés : `Key` et `Compare`. Le type paramétrisé `Compare` est un fonctionnel. La collection `set` réalise la notion mathématique d'un ensemble. Cependant, les éléments d'un `set` sont toujours triés. Tout comme un ensemble, un `set` n'admet pas d'élément identique. La collection `multiset` est identique à la collection `set`. Cependant, un `multiset` admet des éléments identiques. La plupart des opérations ensemblistes sont disponibles dans ces collections.

Dans les collections `set` et `multiset` le type des éléments est représenté par le type paramétrisé `Key`. Le fonctionnel de type `Compare` est utilisé pour trier les éléments de la collection. Voici un exemple des éléments d'un `set` :

```
111
222
333
444
555
```

Voici un exemple des éléments d'un `multiset` :

```
1234
5678

5678
9998
9999
```

Le programme ci-dessous démontre l'utilisation de la collection `set`.

```
// Exemple d'utilisation d'un set
// Note: 7 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <set>

using namespace std;

```

```
typedef set<int, less<int> >::iterator set_iterateur;

int main()
{
 set_iterateur i;
 set<int, less<int> > s1, s2;

 // set s1
 s1.insert(10);
 s1.insert(20);
 s1.insert(30);

 // set s2
 s2.insert(30);
 s2.insert(10);
 s2.insert(20);

 if (s1 == s2)
  cout << "Ensemble s1 identique a s2" << endl;

 for (i = s1.begin(); i != s1.end(); i++)
  cout << *i << " ";
 cout << endl;

 return (0);

}
```

Le résultat affiché est :

```
Ensemble s1 identique a s2
 30
```

L'utilisation d'un `multiset` est tout à fait analogue à celle d'un `set`.

```
// Exemple d'utilisation d'un multiset
// Note: 9 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <set>

using namespace std;
typedef multiset<int, greater<int> >::iterator multiset_iterateur;


int main()
{
 multiset_iterateur i;

 multiset<int, greater<int> > ms1, ms2;

 ms1.insert(10);
 ms1.insert(20);
 ms1.insert(30);
 ms1.insert(10);

 ms2.insert(10);
 ms2.insert(20);
 ms2.insert(30);

 if (ms1 == ms2)
  cout << "Multiset ms1 identique a ms2\n";
 else
  cout << "Multiset ms1 n'est pas identique a
↳ms2\n";

```

```
cout << "Contenu de ms1:\n";
for (i = ms1.begin(); i != ms1.end(); i++)
 cout << *i << " ";
cout << endl;

cout << "Contenu de ms2:\n";
for (i = ms2.begin(); i != ms2.end(); i++)
 cout << *i << " ";
cout << endl;

return (0);


}
```

Le résultat affiché est :

```
Multiset ms1 n'est pas identique a ms2
Contenu de ms1:
30 20 10 10
Contenu de ms2:
30 20 10
```

2.8.2 MAP ET MULTIMAP

Les collections `map` et `multimap` possèdent trois types paramétrisés : `Key`, `T` et `Compare`. Ce dernier est un fonctionnel. La collection `map` réalise l'association d'une donnée à une clé. Cependant, les éléments d'un `map` sont toujours triés. Tout comme un `set`, un `map` n'admet pas de clé identique.

La collection `multimap` est identique à la collection `map`. Cependant, un `multimap` admet des **clés identiques**. Dans les collections `map` et `multimap` le type des clés est représenté par le type paramétrisé `Key`. Aussi, dans les collections `map` et `multimap` le type des données est représenté par le type paramétrisé `T`. Enfin, le fonctionnel de type `Compare` est utilisé pour trier les clés de ces collections. Voici un exemple des données d'un `map`.

```
(Key T)
111 Jean
222 Michel
333 Jarre
444 Printemps
555 Equinox
```

Voici un exemple des données d'un `multimap` :

```
(Key T)
111 Jean
333 Michel
333 Jarre
444 Printemps
555 Equinox
```

Les données d'un `map` et d'un `multimap` sont encapsulées dans une classe paramétrisée `pair`. ☞

L'ensemble (clé, donnée) d'un `map` et `multimap` est en réalité encapsulé dans une classe paramétrisée appelé `pair`. Le code ci-dessous est un exemple de cette classe paramétrisée fort utile.

```
// Exemple d'utilisation de la classe pair

#include <iostream.h>
#include <utility>

using namespace std;

int main()
{
 pair<double, int> P1(3.1416, 2), P2 = P1;

 P2 = make_pair(6.11, 3);

 cout << "P1: " << P1.first << " " << P1.second << endl;
 cout << "P2: " << P2.first << " " << P2.second << endl;

 if (P2 > P1)
  cout << "P2 > P1\n";

 ++P1.first; ++P1.second;
 cout << "Apres preincrementation: P1 est " <<
  P1.first << " " << P1.second << endl;

 return (0);
}
```

Le résultat affiché est :

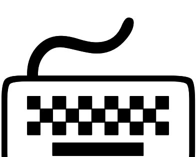
```
P1: 3.1416 2
P2: 6.11 3
P2 > P1
Apres preincrementation: P1 est 4.1416 3
```

Noter bien que la classe paramétrisée `pair` n'a pas de constructeur par défaut dans la définition originale de STL. Donc, en théorie, **on ne peut pas écrire** :

```
pair<double, int> P1;
```

Par contre, il existe un constructeur par défaut dans le Visual C++ !!

Nous allons donner un exemple qui montre bien la puissance des collections associatives. Le code suivant réalise un petit programme de gestion des employés. Plus particulièrement, il donne le salaire des employés. Remarquer bien l'utilisation d'un map dont la clé est une chaîne de caractères. Ainsi, on peut directement associer le nom de l'employé avec son salaire!



```
// Exemple d'utilisation de la classe map
// Note: 19 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <string.h>
#include <map>

using namespace std;

```

```
class CompareChaine {
public:
 bool operator()(const char *str1, const char *str2) const

 {
  // str1 plus petit que str2 ?
  return (strcmp(str1, str2) < 0);
 }
};

typedef map<char*, float, CompareChaine> FicheEmploye;

int main()
{
 FicheEmploye Salaire;
 char Nom[30];

 Salaire["Jean"] = 12000.0f;
 Salaire["Michel"] = 12500.0f;
 Salaire["Jarre"] = 11001.0f;
 Salaire["Paul"] = 5000.0f;

 cout << "Entrer un nom: ";
 cin.get(Nom, 30);

 // Chercher
 if (Salaire.find(Nom) != Salaire.end())
  cout << "Son salaire est: " << Salaire[Nom];
 else
  cout << "Nom pas trouver: " << Nom;

 cout << endl;

 return (0);

}
```

Le résultat affiché est :

```
Entrer un nom: Jean
Son salaire est: 12000
```

Un autre exemple de l'utilité de la collection associative `map`. Cette fois, le map est utilisé comme un tableau dont l'indice est un nombre en point flottant!

```
// Exemple d'utilisation de la classe map (2)
// Note: 19 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <map>

using namespace std;

const float PI = 3.1416f;

typedef map<float, float, less<float> > Lecture;

int main()
{
 Lecture TempsAmplitude;
 Lecture::iterator iter;
 int i;
 float temps = 0.1f, freq = 10.0f;

 for (i=0; i<10; i++)
```

```
  TempsAmplitude[temps*i] = 2*PI*freq*temps*i;

 cout << "Nombre d'association: " <<
  TempsAmplitude.size() << endl;

 for (iter = TempsAmplitude.begin(); iter !=
↳TempsAmplitude.end(); iter++)
  cout << (*iter).first << "\t\t" <<

(*iter).second << endl;

 return (0);

}
```

Le résultat affiché est :

```
Nombre d'association: 10
0               0
0.1             6.2832
0.2             12.5664
0.3             18.8496
0.4             25.1328
0.5             31.416
0.6             37.6992
0.7             43.9824
0.8             50.2656
0.9             56.5488
```

On voit bien qu'il est possible de créer un tableau dont l'indice n'est pas un type intégral. Concentrons-nous maintenant à une application exemple utilisant un `multimap` :

```
// Exemple d'utilisation de la classe multimap
// Note: 19 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <map>
using namespace std;

typedef multimap<float, char*, greater<float> > Pointage;

int main()
{
 Pointage MeilleurPointage;
 Pointage::iterator i;

 MeilleurPointage.insert(Pointage::value_type(12.5f,
↳"Jean"));
 MeilleurPointage.insert(Pointage::value_type(19.15f,
↳"Michel"));
 MeilleurPointage.insert(Pointage::value_type(10.5f,
↳"Tony"));
 MeilleurPointage.insert(Pointage::value_type(15.5f,
↳"Isabelle"));
 MeilleurPointage.insert(Pointage::value_type(12.5f,

↳"Dalida"));
 MeilleurPointage.insert(Pointage::value_type(10.5f,
↳"Pascal"));

 cout << "Meilleur pointage\n";
 cout << "Nombre d'association: " <<
  MeilleurPointage.size() << endl;

```

```
for (i = MeilleurPointage.begin(); i !=

↳MeilleurPointage.end(); i++)
 cout << (*i).first << "\t" << (*i).second << endl;

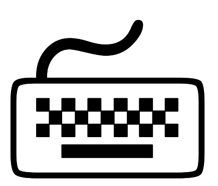
return (0);

}
```

Le résultat affiché est :

```
Meilleur pointage
Nombre d'association: 6
19.15   Michel
15.5    Isabelle
12.5    Jean
12.5    Dalida
10.5    Tony
 Pascal
```

Enfin, nous terminons ce chapitre par un dernier exemple de l'utilisation d'un `multimap` :



```
// Exemple d'utilisation de la classe multimap (2)
// Note: 20 avertissements après compilation configuration
//       debug

#include <iostream.h>
#include <map>

using namespace std;

class CompareChaine {
public:
 bool operator()(const char *str1, const char *str2) const
 {
  // str1 plus petit que str2 ?
  return (strcmp(str1, str2) < 0);
 }
};

typedef multimap<char*, int, CompareChaine> ID;
typedef ID::iterator iterateur;

void Insertion(ID& IDCarte, char *nom, int numero)
{
 iterateur i = IDCarte.insert(ID::value_type(nom,
↳numero));

 cout << (*i).first << "\t\t" << (*i).second <<
↳"\ttraite.\n";

}

int main()
{

 ID Etudiants;
 iterateur i;

 Insertion(Etudiants, "Jean", 101);
 Insertion(Etudiants, "Paul", 102);
 Insertion(Etudiants, "Pierre", 103);
 Insertion(Etudiants, "Jean", 104);
 Insertion(Etudiants, "Michel", 105);
 Insertion(Etudiants, "Luc", 106);
```

```
43. Insertion(Etudiants, "Yannick", 107);
44. Insertion(Etudiants, "Isabelle", 108);
45. Insertion(Etudiants, "Dalida", 109);
46. Insertion(Etudiants, "Isabelle", 110);
47.
48. cout << "\nLe contenu de la table d'identification\n";
49. for (i = Etudiants.begin(); i != Etudiants.end(); i++)
50.  cout << (*i).first << "\t\t" << (*i).second <<
51.↳endl;
52.
53. cout << "\nElimination des etudiants dont le prenom est
54.↳Jean\n";
55. cout << Etudiants.erase("Jean") << " Jean effaces\n";
56.
57. return (0);
58.}
```

Le résultat affiché est :

```
Jean            101     traite.
Paul            102     traite.
Pierre          103     traite.
Jean            104     traite.
Michel          105     traite.
Luc             106     traite.
Yannick         107     traite.
Isabelle        108     traite.
Dalida          109     traite.
Isabelle        110     traite.

Le contenu de la table d'identification
Dalida          109
Isabelle        108
Isabelle        110
Jean            101
Jean            104
Luc             106
Michel          105
Paul            102
Pierre          103
Yannick         107

Elimination des etudiants dont le prenom est Jean
2 Jean effaces
```

Dans cet exemple, nous avons créé une liste d'étudiants utilisant seulement leur prénom et un numéro d'identification. Le prénom est utilisé comme clé dans le `multimap`. Il est donc tout à fait possible de placer des étudiants de même prénom dans cette liste d'étudiants. À la ligne 56 du programme, nous effectuons l'élimination des étudiants portant le prénom « `Jean` ». La fonction membre `erase` d'un multiset élimine automatiquement toutes les données portant la même clé dans la collection.

2.9 CHAÎNES DE CARACTÈRES

Les chaînes de caractères ont leur propre collection dans le STL. En effet, l'efficacité d'un vecteur de caractères ou d'une liste de caractères est douteuse. C'est pour raison que le STL nous offre une spécialisation dans le traitement des chaînes de caractères

sous la forme d'une classe collection `string`[4]. Il existe un nombre impressionnant de fonctions membres et d'opérateurs dans la classe `string`. La plupart des manipulations de chaînes sont déjà implantées dans cette classe. Évidemment, la gestion de la mémoire est également automatisée. Ainsi, nous n'aurons pas à se soucier de la taille des chaînes et l'allocation dynamique de la mémoire pour entreposer ces chaînes de caractères.

Voici un exemple d'utilisation des objets de type `string`. Nous désirons détecter les lettres GPA, ETS et UQ dans une ligne de texte. Cette ligne de texte est saisie directement de l'entrée standard.

```
// Chaînes: Exemple utilisant la classe string
// Trouver des mots clés dans une chaîne

#pragma warning (disable : 4786) // Enlever l'avertissement C4786
#pragma warning (disable : 4503) // Enlever l'avertissement C4503

#include <iostream>          // pour la fonction STL getline
#include <string>

using namespace std;

// Mot clés
static char MOTCLES[][4] = { "GPA", "ETS", "UQ " };

int main(int argc, char* argv[])
{
 cout << "Chaines: Exemple utilisant la classe string" << endl << endl;

 cout << "Je cherche les mots cles: ";
 for (int i=0; i<3; i++) cout << MOTCLES[i] << ' ';
 cout << endl << "Entrer une chaine de caracteres: " << endl;
 // IMPORTANT !!!
 // Il y a une erreur dans l'implantation Microsoft de la fonction membre
 // getline(). Voir:
 //   http://support.microsoft.com/support/kb/articles/q240/0/15.asp
 // pour connaître la façon de corriger cette erreur.
 // Ce n'est pas une erreur de STL mais bien une erreur d'implantation de
 // Microsoft VC++.
 string userinput;
 getline(cin, userinput);

 if (!userinput.empty()) {
  cout << "Nombre de caracteres dans la chaine: " << userinput.length() <<
  endl;
  int index;
  for (i=0; i<3; i++) {
   // Trouver les mots clés dans la phrase
   index = userinput.find(MOTCLES[i][0], 0);
   if (index >= 0)
    cout << "Mot cle: " << MOTCLES[i] << " trouve !" << endl;
  }
 }
 else

  cout << "Chaine vide." << endl;

 return 0;
}

```

[4] `string` représente une chaîne de caractères ANSI, `wstring` représente une chaîne de caractères en UNICODE.

La fonction STL à la ligne 30 effectue la lecture des caractères à partir de l'entrée standard (`cin`). Le nombre de caractères lu n'a pas d'importance puisque la variable `userinput` de type `string` gère sa propre mémoire.

Enfin, les lignes 36 à 41 réalisent la fouille des mots clés dans la ligne de texte à l'aide de la fonction membre `string::find()`. La réalisation de ce code est très simple et ne nécessite que quelques lignes de programmation. En effet, les lignes de commentaires sont presque aussi nombreuses que les lignes de code !

2.10 EXEMPLES DE PROGRAMMATION

Cette section présente un ensemble de programmes de démonstration. Ces programmes utilisent les services offerts par la bibliothèque STL. Voici une brève description de ces programmes exemples.

Nom	Description
`Chaines`	Manipulation des chaînes de caractères par des objets de type `string`.
`Distance`	Parcours de graphe, algorithme de Dijkstra.
`Eratosthenes`	Trouver les nombres premiers contenus dans 1 à N. Méthode naïve.
`Inventaire`	Petit système d'inventaire. Utilisation des objets de type `list`.
`Mots`	Montrer la position des mots dans un texte. Utilisation de `map` et `multimap`.
`TPhone`	Petite base de données pour numéro de téléphone.
`Triage`	Triage par radix sort. Utilisation de `deque`.
`FilePrio`	File de priorité.
`Auto1D`	Automate cellulaire 1D.

Tableau 13 Brève description des programmes exemples.

Les projets de programmation de ces exemples sont disponibles sur la page Web de ce cours.

LECTURE SUGGÉRÉE

La bibliothèque STL est très riche. Nous avons effectué un survol de quelques éléments intéressants. Il existe un nombre de composants et outils de STL qui ne sont pas abordés dans cette introduction. Ceux qui sont intéressés par le STL peuvent consulter :

[AMME97] Ammeraal, Leen, *STL for C++ programers*, John Wiley & Sons, 1997.

[MUSS95] Musser, D. R., Saini, A., *STL Tutorial and Reference Guide*, Reading, Addison Wesley, 1995.

[PLAU95] Plauger, P. J., *The Standard Template Library*, C/C++ Users Journal à partir de décembre 1995.

PROBLÈMES

** 2.1 Expliquer le rôle de l'instruction `namespace`. Montrer comment utiliser plus d'un `namespace` dans un programme.

2.2 Créer une fonction paramétrisée qui calcule la moyenne d'un ensemble de données. Le résultat doit être du même type que les données.

2.3 Créer un programme capable de calculer la moyenne d'un ensemble de données. Le résultat affiché doit être du même type que les données. Les données doivent être lues du clavier.

2.4 Même que la question 2.3 cependant l'affichage du résultat doit être réalisé par un itérateur de flux.

2.5 Créer une petite base de données de fiches d'employés. Chaque fiche d'employé comprend : *i)* Nom, *ii)* Adresse; *iii)* NAS; *iv)* Date de naissance; *v)* Poste d'emploi; *vi)* Salaire. Utiliser un multiset pour cette réalisation. Le nom de l'employé jouera le rôle de clé dans le multiset. Réaliser les fonctions `insertion`, `élimination`, `affichage` et `recherche` pour cette base de données.

2.6 Associer la base de données de la question 2.5 à une autre base de données indiquant le nombre d'absence de l'employé (en jours) et les raisons de ces absences (en texte).

2.7 Ajouter les fonctions `triage` (par nom des employés), `jours_absence` (de tous les employés) et `salaire_total` dans le prgramme de la question 2.6.

2.8 Écrire un programme C++ réalisant une calculatrice RPN (notation polonaise inverse) acceptant des nombres quelconques (i.e. entier, point-flottant, imaginaire, etc.). Les opérateurs de cette calculatrice sont {+, -, /, *}.

2.9 Reprendre le programme réalisé en 2.8. Ajouter la gestion d'exception `try-throw-catch` pour gérer la division par zéro et les erreurs d'expression RPN.

Note : Toutes les réalisations doivent utiliser les classes collection de STL.

CHAPITRE 3

Introduction à MFC

Plusieurs réviseurs m'ont demandé de comparer C++ à un autre langage. J'ai décidé de ne pas la faire [cette comparaison] …Je n'ai pas le temps et en tant que concepteur de C++, mon impartialité ne serait jamais complètement crédible.

— *Bjarne Stroustrup, The Design and Evolution of C++.*

Il a été reconnu que la programmation Windows (ou tout autre environnement événementiel) nécessite un grand effort d'apprentissage. D'abord, le paradigme de programmation n'est pas le même que celui utilisé dans la programmation traditionnelle. Dans un environnement événementiel (*event-driven*), le programme droit traiter un ensemble d'événements synchrone et asynchrone pour réaliser son travail. Il n'y a pas de boucle principale qui coordonne toutes les activités du programme comme c'est le cas dans la programmation traditionnelle. Le second facteur qui rend la programmation Windows si ardue est l'envergure monstrueuse de l'interface de programmation (API) du système. La quantité de fonctions disponibles dans l'API de Windows peut dérouter les programmeurs les plus expérimentés. Le MFC peut régler, en partie, les difficultés de la programmation Windows. L'organisation des fonctions de Windows en classes hiérarchisées permet aux programmeurs d'avoir une vue plus consistante et plus logique de l'API. Cette encapsulation de l'API offre également une plus grande souplesse dans le développement des programmes en isolant le programmeur de la mécanique interne de Windows. Cependant, le MFC ne peut pas faire disparaître toutes les difficultés reliées à la programmation Windows. Les développeurs doivent encore apprendre la programmation événementielle !

3. PHILOSOPHIE ET CONCEPTS DE BASE

La programmation Windows est une programmation basée sur des événements. La figure 1 explique ce type de programmation. Dans ce schème, les événements sont générés par le système Windows en réponse à des actions. Ces actions peuvent être initiées par l'utilisateur (i.e. par le clavier et/ou la souris) ou par un fonctionnement interne du système (i.e. par une alarme, par la transmission d'un message, etc.). Ces événements sont ensuite prétraités par le système Windows et envoyés aux applications sous forme de messages.

C'est pour cette raison que chaque application Windows doit posséder dans sa structure des gestionnaires d'événement. Ces gestionnaires ont pour rôle de répondre aux messages reçus et d'appliquer la logique du programme pour réaliser les tâches spécifiques. D'après la Figure 8, il est clair qu'une application Windows n'est pas coordonnée par une boucle de contrôle puisque les événements ne sont pas nécessairement synchronisés dans le temps.

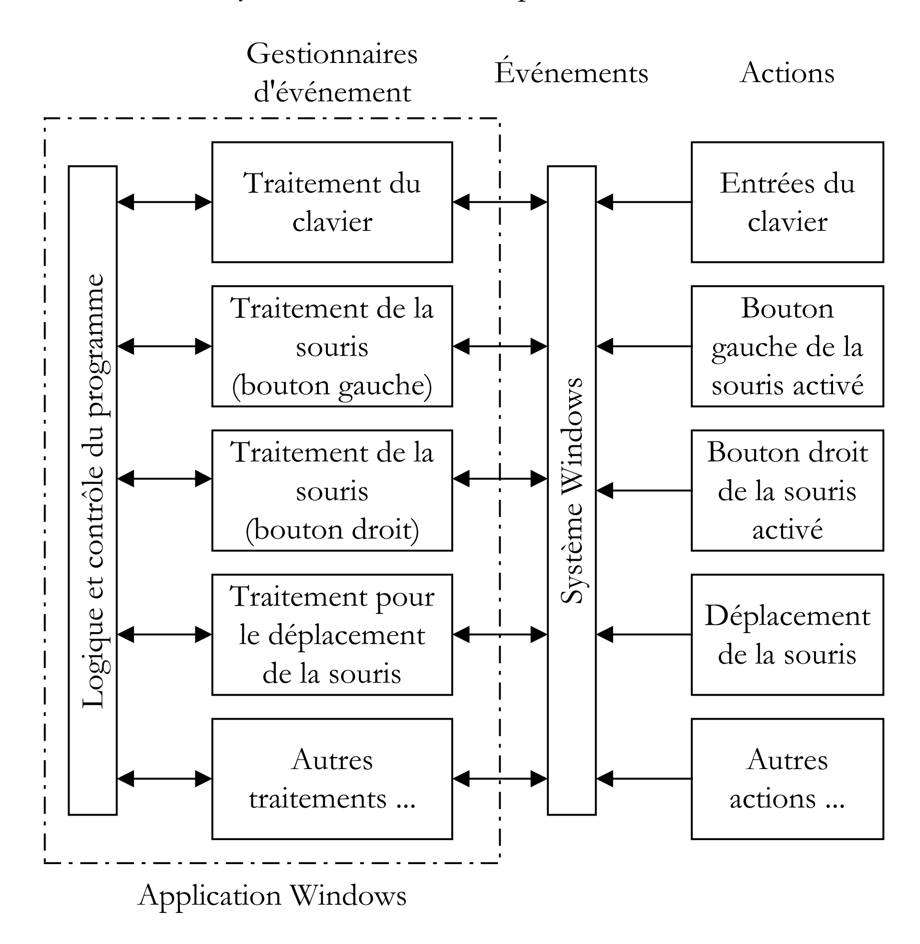
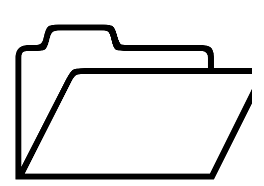


Figure 8 Schéma simplifié montrant le concept de programmation Windows.



Nous pouvons remarquer qu'il existe un gestionnaire pour chaque type de message reçu. Cette remarque est plutôt simplificatrice mais elle est juste dans la majorité des cas. L'astuce est de *faire travailler* ces gestionnaires en harmonie avec le MFC afin d'accomplir une tâche spécifique. Par exemple, pour réaliser l'illusion de déplacement par la souris d'un élément dessiné à l'écran, nous devons traiter le message correspondant à la sélection de l'élément par le bouton gauche enfoncé (message `WM_LBUTTONDOWN`), vérifier la position de la souris à la réception du message (paramètre `CPoint`). Nous devons également obtenir l'élément sélectionné en fouillant les éléments dans la banque de données de l'application (fonction membre `PtInRect`). Si la fouille est positive, l'application doit se mettre en mode de déplacement. À la réception des messages de déplacement de la souris (message `WM_MOUSEMOVE`), le gestionnaire doit indiquer à l'application que l'élément sélectionné a été déplacé. L'application réalise alors l'algorithme d'affichage en mode

NOTXOR (OU-Exclusif inversé) pour effacer l'élément de l'ancienne position et le réafficher à la nouvelle position. Le mode de déplacement restera en vigueur jusqu'à la réception du message de relâchement du bouton gauche de la souris (message WM_LBUTTONUP). Donc, pour réaliser le déplacement d'un élément par la souris, trois messages sont traités. D'où l'importance de bien gérer les messages.

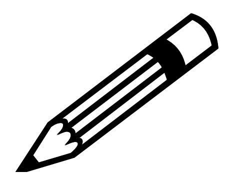
3.1 EXPÉDITION ET ROUTAGE DES MESSAGES

La programmation Windows par le MFC consiste à créer et utiliser des objets dérivés des classes de ce cadre de travail (*Framework*). L'apport le plus important de ce cadre de travail est sans doute l'abstraction du mécanisme de transmission et la réception des messages en provenance de Windows. Sans l'aide de cette abstraction, une énorme boucle de sélection (switch — case) serait nécessaire pour démêler les messages reçus et l'expédier aux gestionnaires de ces messages.

Dans le Visual C++, le problème d'expédition et de routage des messages est résolu en utilisant l'encapsulation de l'API de Windows par le MFC et en établissant le concept de **Cartes de messages** avec la complicité du compilateur. Pour la transmission des messages, les fonctions usuelles de Windows (SendMessage, PostMessage, SendNotifyMessage, etc.) sont disponibles mais elles sont enrobées d'une couche C++ qui simplifie leur utilisation.

Quant au routage des messages, une carte de messages est utilisée. Chacune des classes MFC qui peut recevoir des messages doit posséder une telle carte. En termes simples, une carte de messages est une association directe entre un type de message et une fonction membre qui joue le rôle de gestionnaire. Cette assignation message — gestionnaire n'est pas une caractéristique du langage C++. Elle est réalisée par l'entremise du compilateur à l'aide des macros de C/C++. Dans le MFC, une carte de messages est déclarée dans le code source par l'AppWizard en utilisant le macro suivant :

```
DECLARE_MESSAGE_MAP()
```



Remarquer qu'il n'y a pas de ponctuation à la fin de la ligne. Le macro DECLARE_MESSAGE_MAP() est normalement placé à la fin de la déclaration d'une classe à l'intérieur d'un fichier d'en-tête (.h). La déclaration d'une carte de messages permet au compilateur de générer le code nécessaire pour le routage des messages Windows. La contrepartie de la déclaration de la carte de messages est la définition même de la carte. Cette définition est normalement réalisée dans le fichier source contenant la définition des classes (.cpp). Les macros Visual C++ à utiliser sont :

```
BEGIN_MESSAGE_MAP( CMyWindow, CFrameWnd )
    //{{AFX_MSG_MAP( CMyWindow )
    ON_WM_PAINT()
    ON_COMMAND( IDM_ABOUT, OnAbout )
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

Le macro `BEGIN_MESSAGE_MAP()` sert à indiquer au compilateur que les lignes suivantes sont des associations message — gestionnaire d'une carte de messages. Les lignes `//{{AFX_MSG_MAP( CMyWindow )` et `//}}AFX_MSG_MAP` sont des commentaires spéciaux nécessaires pour le bon fonctionnement de l'outil ClassWizard (voir le document en-ligne de VC++ pour connaître les détails de cet outil). Les lignes contenant les macros `ON_WM_PAINT()` et `ON_COMMAND()` réalisent l'association proprement dite des messages et des gestionnaires. Par exemple, le macro `ON_WM_PAINT()` associe le message `WM_PAINT` avec la fonction membre virtuelle `OnPaint()` de la classe `CWnd` de MFC. Le macro `ON_COMMAND()` associe un identificateur à une fonction membre d'une classe quelconque. Cette dernière macro est utilisée pour associer une option du menu de l'application à une fonction membre d'une classe. Enfin, le macro `END_MESSAGE_MAP()` indique au compilateur que la définition de la carte de messages est terminée.

Il existe une panoplie de macros pour effectuer l'association message — gestionnaire dans Visual C++. Cependant, il n'est pas nécessaire de se souvenir de tous ces macros. Dans l'IDE (*Integrated Development Environment*) de Visual C++, vous pouvez demander à l'outil ClassWizard de mettre en place la carte de messages automatiquement. La Figure 9 montre un exemple de son utilisation.

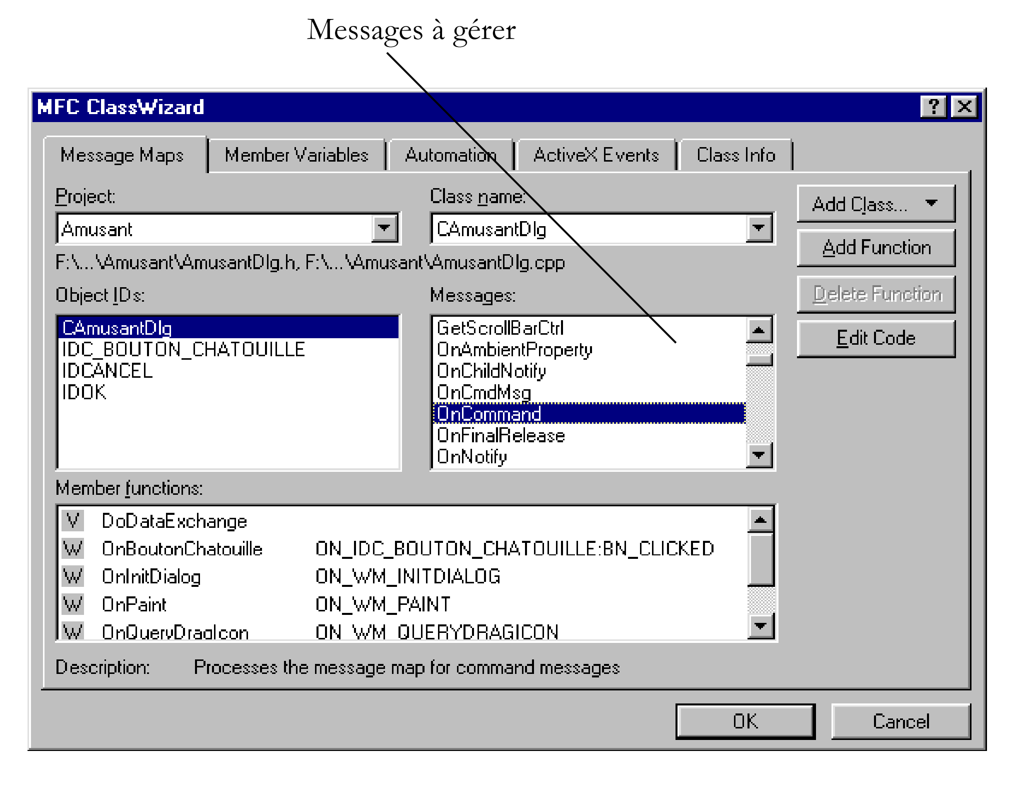
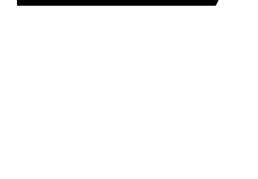


Figure 9 Réalisation d'une association message - gestionnaire par le ClassWizard.

Le fait de sélectionner un message à traiter dans ClassWizard provoque automatiquement l'instauration d'une carte de messages dans la classe. Le ClassWizard ajoutera aussi les macros appropriés pour la définition des associations message — gestionnaire.

3.2 CLASSES IMPORTANTES DU MFC

La hiérarchie des classes de base de MFC est fort complexe. Il n'est pas possible d'expliquer ici toutes ces classes. Nous allons plutôt présenter les classes de base nécessaires pour la réalisation d'une application Windows.



À noter que l'utilisation des classes de MFC implique l'inclusion du fichier d'en-tête `afxwin.h` :

```
#include <afxwin.h>
```

Les fichiers sources générés par AppWizard auront déjà `afxwin.h` appliqués. Cependant, vous devez l'inclure dans vos propres fichiers sources créés sans l'aide de AppWizard.

3.2.1 CLASSE COBJET

C'est la classe de base abstraite ultime dans le MFC. `CObjet` ne contient pas de fonctionnalités Windows proprement dites. Cette classe de base renferme plutôt tous les services nécessaires pour les besoins de la programmation. Par exemple, `CObjet` contient les méthodes pour la sérialisation (entrée/sortie des objets sur disque et dans le presse-papiers), les méthodes pour l'entreposage des objets dans des Collections qui sont ordonnées. Il faut dériver nos classes de `CObjet` pour hériter ces services fort utiles. D'ailleurs la plupart des classes de MFC sont dérivées de cette classe de base abstraite.

3.2.2 CLASSE APPLICATION

La classe d'application `CWinApp` est une classe fondamentale dans la programmation Windows par MFC. Un objet de cette classe réalise toutes les tâches nécessaires pour la mise en place, l'exécution et la terminaison d'une application sous Windows. Donc, il est impératif de dériver une classe à partir de `CWinApp` taillée pour nos besoins. Heureusement, l'utilisation de AppWizard rend ce travail superflu. En effet, l'AppWizard crée automatiquement une classe dérivée de `CWinApp` pour toute nouvelle application. Voici le code généré par AppWizard pour la déclaration d'une classe d'application nommée `CGPA789App` dérivée de `CWinApp`.

```
class CGPA789App : public CWinApp
{
public:
 CGPA789App();

// Overrides
 // ClassWizard generated virtual function overrides
 //{{AFX_VIRTUAL(CGPA789App)
 public:
 virtual BOOL InitInstance();
 //}}AFX_VIRTUAL

// Implementation

//{{AFX_MSG(CGPA789App)
```

```
afx_msg void OnAppAbout();
 // NOTE - the ClassWizard will add and remove
↳ member functions here.
 //    DO NOT EDIT what you see in these blocks
↳ of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

Rappelons-nous que la convention de MFC ajoute le caractère majuscule C devant les noms des classes. La classe CGPA789App ne contient pas un grand nombre de définitions. Elle possède un constructeur[1] CGPA789::CGPA789(), une fonction membre virtuelle InitInstance(), une autre fonction membre OnAppAbout() et une carte de messages.

Le constructeur de la classe CGPA789 n'est pas vraiment nécessaire puisque le compilateur fournit toujours un constructeur par défaut à toutes les classes. Puisque le constructeur de la classe CGPA789 n'est pas obligatoire, il est clair que le vrai travail est accompli par la fonction membre virtuelle InitInstance().C'est dans cette fonction membre que les réglages systèmes sont réalisés.

```
/////////////////////////////////////////////////////
// CGPA789App initialization

BOOL CGPA789App::InitInstance()
{
 AfxEnableControlContainer();

#ifdef _AFXDLL
 Enable3dControls();                // Call this
↳ when using MFC in a shared DLL
#else
 Enable3dControlsStatic();   // Call this when linking to
↳ MFC statically
#endif

 SetRegistryKey(_T("Local AppWizard-Generated
↳ Applications"));

 LoadStdProfileSettings();  // Load standard INI file
↳ options (including MRU)

 // Register the application's document templates.
↳ Document templates
 //  serve as the connection between documents, frame
↳ windows and views.

 CMultiDocTemplate* pDocTemplate;
 pDocTemplate = new CMultiDocTemplate(
  IDR_DESSINTYPE,
  RUNTIME_CLASS(CGPA789Doc),
  RUNTIME_CLASS(CChildFrame), // custom MDI child
↳ frame
  RUNTIME_CLASS(CGPA789View));
 AddDocTemplate(pDocTemplate);

 // create main MDI Frame window
 CMainFrame* pMainFrame = new CMainFrame;

 if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
```

[1] La sémantique du langage C++ sera expliquée dans un autre document.

```
 return FALSE;
m_pMainWnd = pMainFrame;

// Enable drag/drop open
m_pMainWnd->DragAcceptFiles();
// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);

// Parse command line for standard shell commands,
↳ DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
 return FALSE;

// The main window has been initialized, so show and
↳ update it.
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();

return TRUE;
}
```

La fonction virtuelle `InitInstance()` est plutôt complexe. Dans l'exemple ci-dessus, elle active le sous-système OLE (*Object Linking and Embeding*) pour l'application à l'aide de la méthode `AfxEnableControlContainer()`. Elle active aussi le sous-système de l'interface graphique pour donner l'apparence 3-D des contrôles (i.e. boutons, menus, etc.). L'instauration des paramètres de l'application dans le registre de Windows[2] et la préparation d'un document de base pour le mécanisme DOC/VIEW sont aussi réalisées.

Enfin, `InitInstance()` crée un cadre MDI (*Multiple Document Interface*) et l'affiche à l'écran. Un cadre MDI (Classe `CMainFrame`) est l'objet qui représente la fonctionnalité de la fenêtre principale de l'application. Cette fenêtre principale peut contenir d'autres fenêtres à l'intérieur de son espace visible. Il est important de distinguer ici l'objet cadre principal de sa représentation graphique. **L'objet cadre possède les services nécessaires pour gérer les menus et les fenêtres associées à l'application. Par contre, l'objet cadre n'est pas l'encadrement graphique qui est affiché à l'écran**. La représentation graphique (bordure, couleur de fond, etc.) est encapsulée dans l'objet cadre. C'est pour cette raison qu'il est nécessaire de faire appel à l'objet cadre pour qu'il dessine sa représentation graphique à l'écran. En d'autres mots, un objet dérivé de `CMainFrame` possède dans sa description toutes les méthodes nécessaires pour dessiner sa représentation graphique. Les fonctions membres suivantes réalisent le traçage du cadre principal à l'écran.

```
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
```

[2] Le registre de Windows est une base de données qui sert à entreposer les paramètres internes des programmes. Ainsi, son utilisation élimine la nécessité des fichiers `.ini`.

La variable `pMainFrame` est un pointeur sur l'objet cadre et les fonctions membres `ShowWindow()` et `UpdateWindow()` permettent le déclenchement de l'affichage du cadre.

3.2.3 CLASSE FENÊTRE MDI

Une fenêtre est un espace délimité dans lequel les données sont affichées et où l'utilisateur peut interagir avec le contenu d'une application. Dans une application MDI, chacune des fenêtres est une enfant du cadre principal. Cette relation est logique puisque les fenêtres créées dans une application MDI partagent le même menu principal. Un exemple concret est l'application Word de Microsoft. Il est possible d'éditer plusieurs fichiers dans Word (à l'intérieur de fenêtres différentes) mais il n'existe qu'un seul menu dans le cadre principal. La Figure 10 montre ce principe.

La classe MFC qui encapsule les enfants fenêtres MDI est `CMDIChildWnd`. L'affichage graphique des fenêtres enfants de la classe `CMDIChildWnd` peut être créée de trois façons. Les deux premières façons consistent à effectuer la création directement par les fonctions membres `Create()` et `LoadFrame()`. La troisième façon exige la collaboration du mécanisme DOC/VIEW.

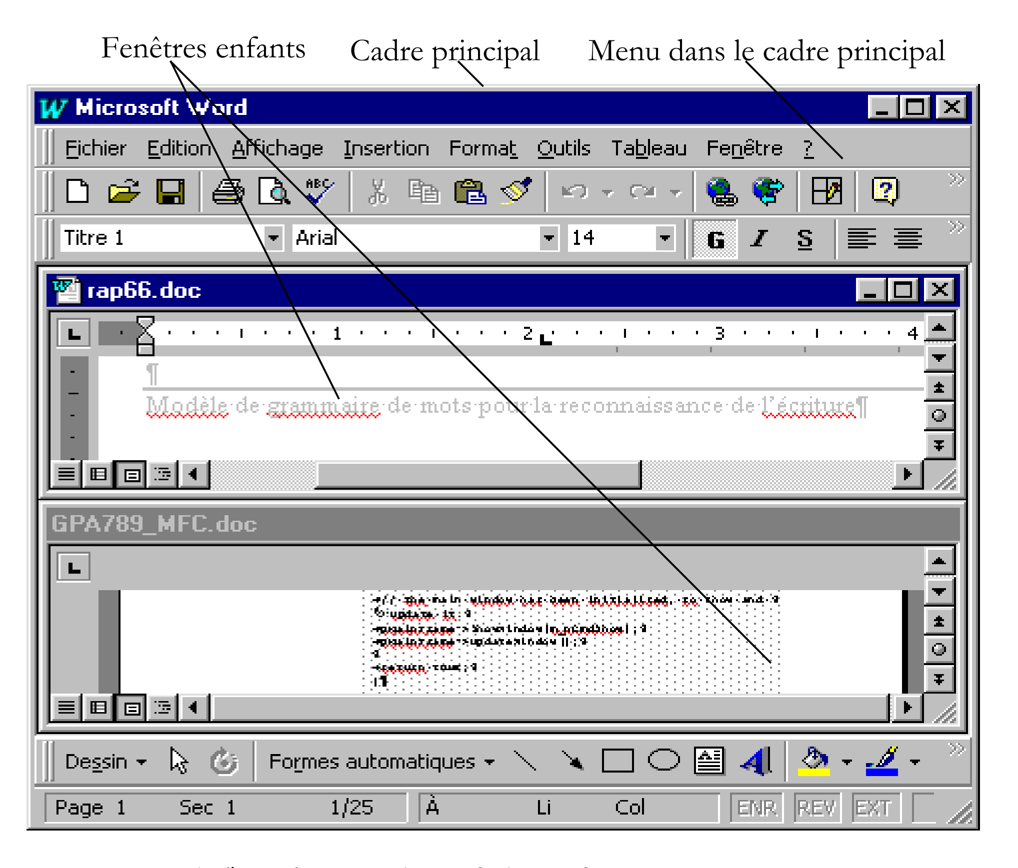


Figure 10 Exemple d'un cadre principal et ses fenêtres enfants.

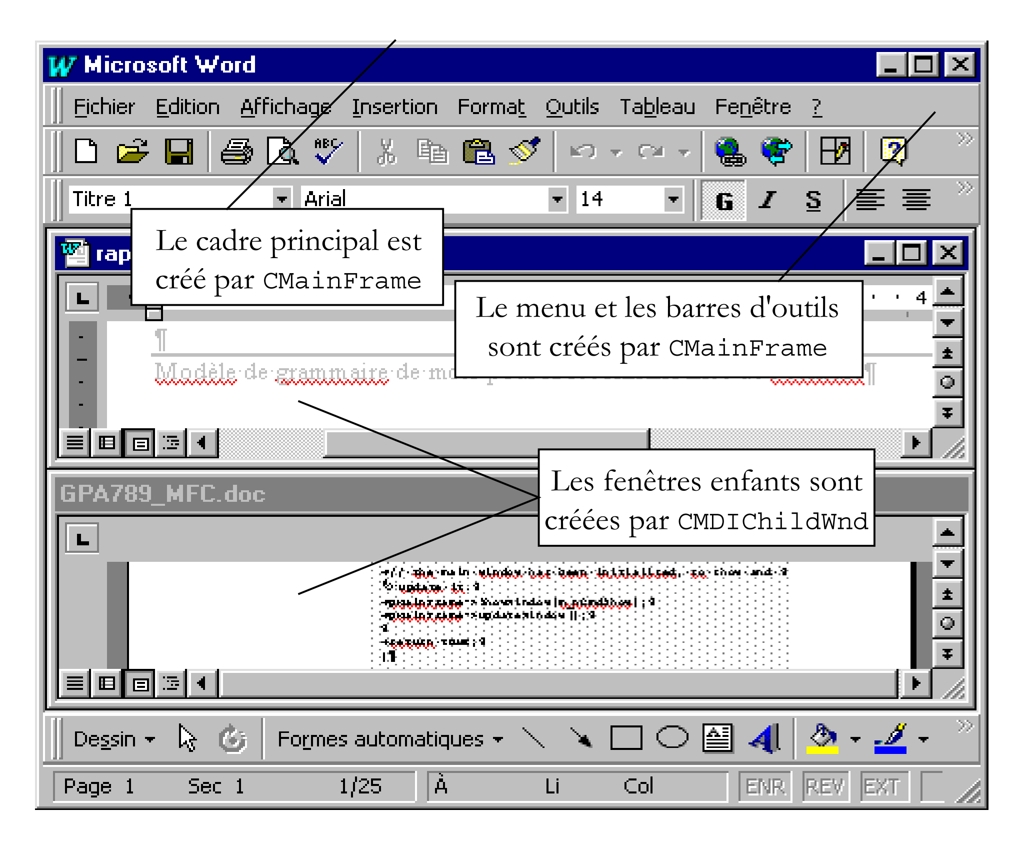


Figure 11 Classes de base associées aux éléments d'interface.

La Figure 11 énumère les différentes classes de MFC correspondant aux éléments d'interface graphiques. Rappelons-nous encore une fois que les classes MFC **ne sont pas** des éléments d'interface affichés. Par contre, elles possèdent les méthodes nécessaires pour la création et l'affichage des éléments d'interface.

3.2.4 CLASSES COLLECTIONS

Le MFC ne contient pas uniquement les classes encapsulant les éléments graphiques de Windows. Il renferme également un ensemble de classes qui servent à regrouper des objets dans des structures de données. Une collection est une agrégation d'articles qui sont organisés selon une façon déterminée. Par exemple, une pile (*stack*) organise les articles selon le schème premier entré dernier sorti (FILO). De plus, une pile possède un nombre de règles qui décrit exactement son fonctionnement. Les actions possibles d'une pile sont : empiler, dépiler, vider, etc.. On remarquera que la réalisation informatique de la pile n'est pas normalement spécifiée. Ainsi, il est possible de réaliser une pile à l'aide d'un tableau, d'une liste chaînée ou tout autre construction informatique.

Les classes Collections de MFC réalisent différentes d'agrégation d'objets en offrant trois types de construction. Ces constructions sont : *i)* tableau, *ii)* liste, *iii)* carte. Un tableau est semblable à un vecteur de STL (*Standard Template Library*). On peut indexer dans le tableau à l'aide d'un indice numérique. La taille d'un tableau est dynamique et la gestion de la mémoire est réalisée directement par le tableau. Une

liste est un arrangement ordonné d'objets semblables à une liste chaînée. Son opération est plus rapide qu'un tableau puisqu'il est possible d'insérer et d'éliminer un objet sans déplacer les objets déjà contenus dans la liste. Enfin, une carte est une collection d'objets sans ordre. Un objet contenu dans une carte est associé à une clé alphanumérique. Son opération est très rapide puisqu'il n'y a pas de fouille à effectuer pour retrouver un objet. L'unicité de la clé associée permet la manipulation directe de l'objet. L'implantation des classes Collections de MFC est basée sur le mécanisme de classes paramétrisées de C++. Par exemple, la classe `CArray` est la réalisation d'un tableau dynamique dans MFC, la classe `CList` est la réalisation d'une liste chaînée (doublement chaînée) et la classe `CMap` est la réalisation d'une carte associative. Toutes ces classes permettent l'entreposage direct des objets. Il existe également des versions de classes Collections qui entreposent des pointeurs d'objets.

3.3 CONCEPT DOC/VIEW

Le cadre de travail MFC impose une structure bien déterminée dans la programmation Windows. Cette structure concerne la façon donc les données de l'application sont entreposées et traitées. Elle impose également deux entités informatiques pour toutes les applications créées. Ces entités informatiques sont le **Document** et la **Vue**.

Deux vues différentes d'un même document (le texte de ce manuel)

Figure 12 Exemple de vues associées à un document.

Un document dans le concept DOC/VIEW est une collection de données dans une application. Cette collection de données peut représenter des valeurs numériques, du texte, des objets, des images, etc. Donc, le terme document dans le sens MFC est beaucoup plus large qu'un simple fichier de texte. Un document est créé à partir de la classe `CDocument`. C'est la responsabilité du programmeur d'insérer les services nécessaires pour manipuler les données contenues dans le document. Par exemple, nous pouvons ajouter une classe Collections `CListe` dans la classe document d'un logiciel de dessin. Dans le MFC, il n'est pas nécessaire d'avoir un seul document par application. D'ailleurs, la Figure 11 et Figure 12 montrent très bien la possibilité d'avoir plus d'un document (collection de données) en même temps. L'existence de document multiple dans une application demande une interface particulière. Cette interface est le MDI. Le *Multiple Document Interface* permet l'existence, dans une même application, de plusieurs documents au sens MFC du terme.

La contrepartie des documents dans le concept DOC/VIEW est une vue. Une vue est une entité informatique qui est toujours reliée à un document. Elle est un mécanisme qui permet d'afficher dans une fenêtre enfant Windows, en tout ou en partie, les données contenues dans un document. Dans une interface MDI, on peut avoir plusieurs vues d'un même document. Chacune des vues peut avoir une présentation différente du document. La Figure 12 est un exemple de vues multiples pour un même document. En utilisant l'interface MDI, le concept DOC/VIEW permet également d'associer des vues à des documents différents. Cette association multi-document est montrée dans la Figure 13.

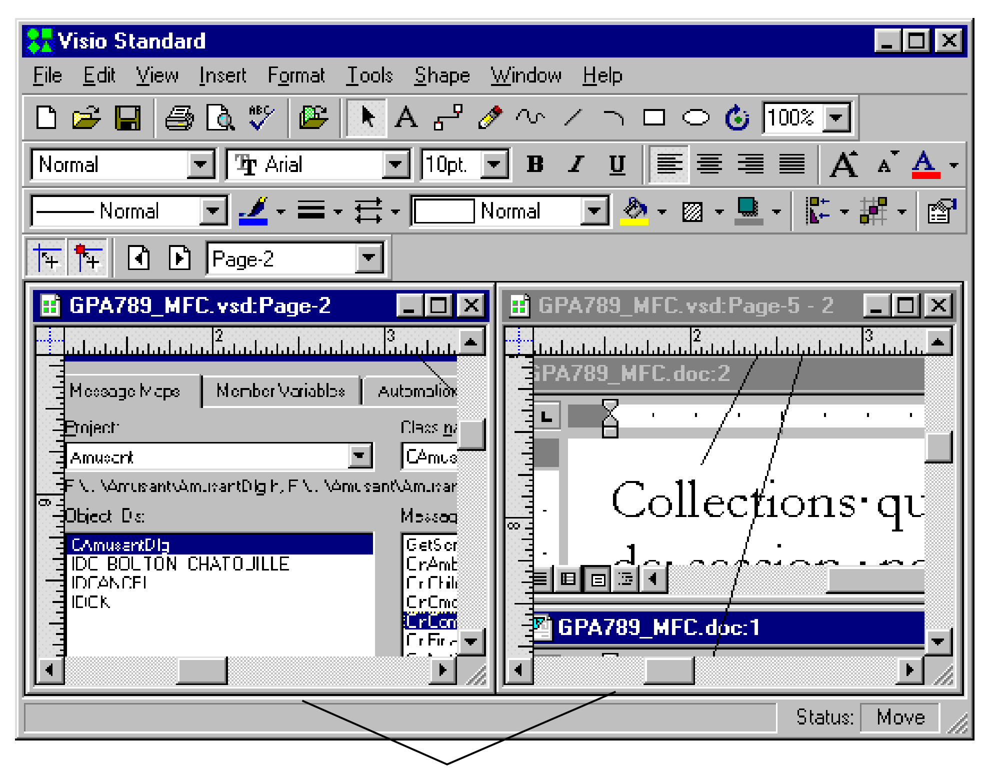


Figure 13 Exemple de document multiple.

3.3.1 LIAISON D'UN DOCUMENT ET SES VUES

L'intégration d'un document, ses vues et les fenêtres enfants est réalisée d'une manière très élégante dans le MFC. Très simplement, un objet dérivé de `CDocument` maintient une liste de pointeurs vers les vues et un objet dérivé de `CView` contient un pointeur vers le document associé. Chacune des fenêtres enfants utilisées pour l'affichage possède aussi un pointeur à l'objet d'une vue active. Enfin, tous ces objets sont synchronisés par un autre objet de MFC issu de la classe *Document template* (classe `CMultiDocTemplate`).

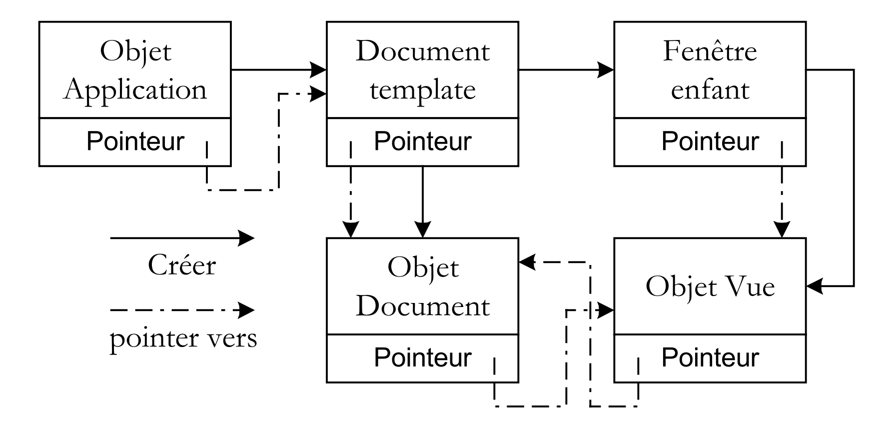


Figure 14 Relations entre les objets importants d'un document et ses vues.

C'est la classe `CMultiDocTemplate` qui gère les objets documents, les objets vues et les fenêtres enfants de l'application. La signification de template en C++ a été abordée dans les classes Collections. La même notion est utilisée ici dans un document template. La Figure 14 donne une représentation graphique de cette liaison document — Vues à l'aide d'un document template. Selon le diagramme de la Figure 14, un objet de la classe `CMultiDocTemplate` est responsable de la création d'un objet document et d'une fenêtre enfant. Cette dernière a, quant à elle, la responsabilité de créer une vue du document.

La classe `CMultiDocTemplate` est utilisée dans la fonction membre `InitInstance()` de la classe Application. Le code ci-dessous est tiré de l'exemple donné au début de ce chapitre.

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
 IDR_DESSINTYPE,
 RUNTIME_CLASS(CGPA789Doc),
 RUNTIME_CLASS(CChildFrame), // custom MDI child frame
 RUNTIME_CLASS(CGPA789View));
AddDocTemplate(pDocTemplate);
```

Dans ces lignes, un objet de la classe `CMultiDocTemplate` est créé dynamiquement et enregistré dans l'application par `AddDocTemplate()` une fonction membre de la classe `CWinApp`. On peut deviner facilement qu'il doit exister un objet `CMultiDocTemplate` par type de document d'où la nécessité d'enregistrer l'objet dans l'application. S'il n'y a qu'un seul type de document alors, un seul objet

`CMultiDocTemplate` est nécessaire et un seul enregistrement dans la fonction `InitInstance()`.

3.3.2 HIÉRARCHIE DES CLASSES

Le MFC contient un très grand nombre de classes. Mais chose surprenante, il est possible de produire une application non triviale en ne faisant appel qu'à un petit nombre de classes. Figure 15 est un graphe hiérarchique des classes MFC normalement impliquées dans la création d'une application. Le rôle principal des classes est également indiqué.

La classe abstraite de base du tout MFC est la classe `CObjet`. `CObjet` est doté de tous les services supports pour la sérialisation et l'entreposage des objets dérivés dans les collections. La classe `CObjet` est l'ancêtre de toutes les classes Windows de MFC.

Une autre classe abstraite dérivée directement de `CObjet` est la classe `CCmdTarget`. Elle offre le mécanisme de routage de messages pour les sélections de menus et barres d'outils (*Commands*) vers les vues et les documents (*Targets*). Toutes les classes qui peuvent recevoir des messages sont dérivées de `CCmdTarget`.

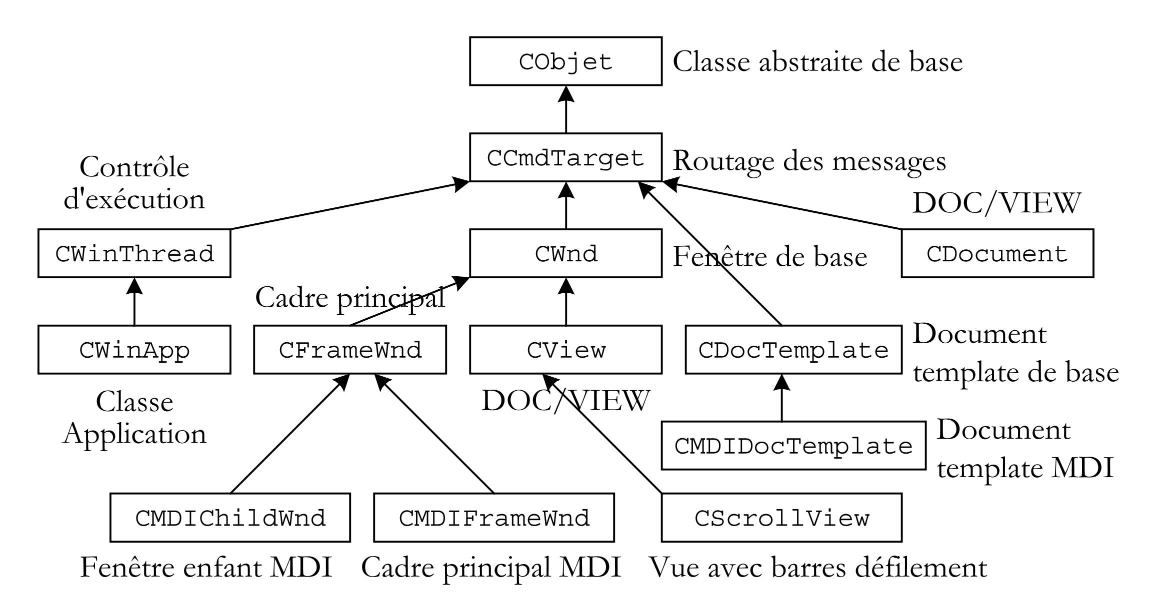


Figure 15 Hiérarchie des classes de base constituant une application.

Dans le MFC, l'objet Application (`CWinApp`) est hérité d'un objet fil d'exécution (*Control Thread*) de la classe `CWinThread`. Un fil d'exécution est une version moderne du concept de coroutine. Il s'agit d'un segment du programme qui s'exécute indépendamment mais partageant le même espace d'exécution que le programme principal. Un fil d'exécution est donc plus simple à créer et la complexité de sa gestion par le système d'exploitation est aussi réduite.

Il existe deux types de fil d'exécution dans le MFC : *i*) fil de travail (*Worker Thread*); *ii*) fil d'interface (*Interface Thread*). Un fil de travail est utilisé pour des tâches qui n'impliquent pas la réception de messages. Par exemple, la comptabilisation en arrière plan du nombre total de requêtes journalières d'une base de transaction peut être réalisée dans un fil de travail. Quant au fil d'interface, il s'agit d'un fil de contrôle qui peut recevoir des messages Windows. Un objet de la classe `CWinApp` est justement un fil de type interface. Il existe donc au moins un fil de contrôle dans toute application MFC. Le concept de fil de contrôle ne sera pas abordé dans le cadre de ce chapitre.

Les classes `CMDIFrameWnd` et `CMDIChildWnd` sont des descendantes de la classe `CWnd`. Cette dernière possède tous les services nécessaires pour la gestion des fenêtres à l'intérieur de MFC. La classe `CMDIFrameWnd` est une spécialisation de `CWnd` et elle réalise le cadre principal d'une application et gère l'emplacement des menus, des barres d'outils et de la barre d'état. De même, la classe `CMDIChildWnd` est une spécialisation de la classe de base `CWnd` et offre les services d'une fenêtre enfant dans une application utilisant l'interface MDI.

Comme nous l'avons vu précédemment, les classes `CDocument` et `CView` servent à supporter le concept DOC/VIEW de MFC. La classe `CView`, bien qu'elles ne soient pas montrées dans l'hiérarchie de la Figure 15, possède 10 spécialisations. La spécialisation `CScrollView`, la seule montrée dans la Figure 15, est une classe dérivée de `CView` mais dotée de services pour la gestion des barres de défilement dans la vue.

Enfin, pour réaliser le concept DOC/VIEW, nous avons besoin d'une descendante de la classe `CDocTemplate`. Rappelons qu'un template de document sert à effectuer la liaison entre un document et ses vues. Pour une application MDI, la classe gestionnaire est la `CMDIDocTemplate`.

À noter que ces classes sont normalement utilisées par AppWizard dans la génération du code source initial. Donc, vous pouvez consulter le code source généré pour connaître le mode d'emploi de ces classes.

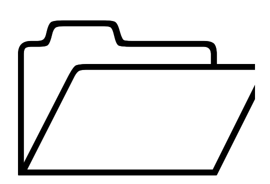
3.3.3 AFFICHAGE SOUS MFC

La capacité d'affichage est fondamentale dans Windows. Après tout, Windows est un système d'interface (Windows NT est aussi un système d'exploitation) qui représente ses éléments sous forme graphique. Pour le sous-système d'affichage, Windows utilise une technique éprouvée qui consiste à rendre les fonctions de dessin indépendantes des fonctions de bas niveau matériel. Le résultat est une interface de programmation (API) consistante tout en laissant une grande lassitude aux fabricants des périphériques d'affichage (i.e. vidéo, imprimante, acétate électronique, etc.) pour la réalisation matérielle et logicielle des fonctions de bas niveau.

C'est à cause de cette indépendance fonctionnelle qu'il est possible d'afficher le même contenu sur des écrans pilotés par des cartes vidéo de différents fabricants. Il

en est de même pour les impressions sur papier. Évidemment, il peut exister des exceptions (le monde des PC!). Un manufacturier de carte vidéo peut très bien offrir des fonctions exceptionnelles pour le rendu d'image. Pour pouvoir profiter de ses capacités extraordinaires, il est nécessaire d'utiliser, dans la programmation, la bibliothèque spéciale de la carte vidéo. D'où la prolifération des SDK (Software Development Kit) dédiés.

Le système Windows offre un grand nombre de fonctions dans son sous-système d'affichage. Toutes ces fonctions sont organisées dans une bibliothèque dynamique `GDI.DLL`. Les fonctions contenues dans le GDI (*Graphics Device Interface*) constituent l'interface de programmation du sous-système d'affichage. Si l'affichage est destiné à l'écran alors les commandes seront transférées au pilote la carte vidéo. Il en est de même pour les imprimantes. Les commandes sont dirigées vers le pilote des imprimantes.



Le MFC offre une encapsulation complète du sous-système d'affichage de Windows. Les fonctions GDI de Windows sont encapsulées dans une classe nommée `CDC`. La classe `CDC` représente également la structure de données DC (Device Context) de Windows. C'est à travers le DC que l'affichage est réalisé. **À noter qu'il n'y a pas d'affichage direct dans la mémoire vidéo ou dans la mémoire de l'imprimante sans l'intervention de Windows**. Ceci est tout à fait normal puisque l'on peut exécuter *p* programme à la fois où $p > 1$. Imaginer les conflits possibles si tous ces programmes manipulent directement la mémoire des périphériques.

Pour donner une idée de l'encapsulation du DC par le MFC, la Figure 16 est un graphe de la hiérarchie montrant les classes dérivées de `CDC`.

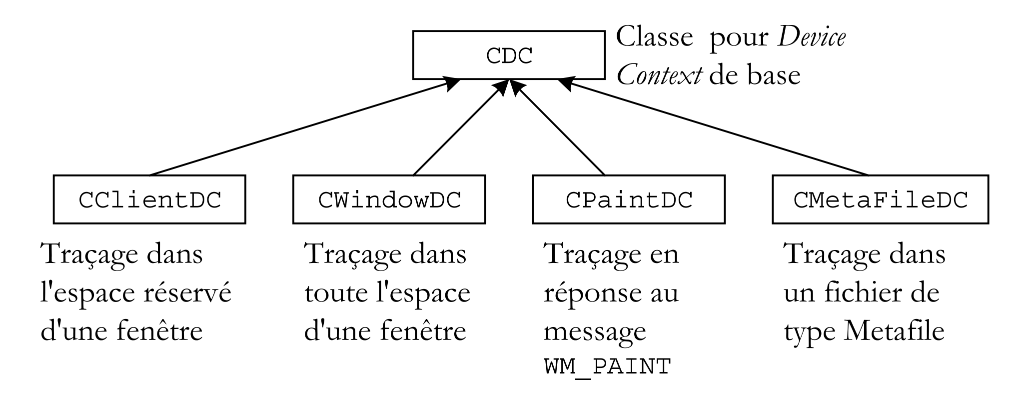


Figure 16 Hiérarchie des classes pour le DC.

La classe `CDC` n'est pas une classe abstraite. Il est donc possible d'instancier directement un objet à partir de `CDC`. La classe `CDC` permet l'affichage sur le bureau (*Desktop*) de Windows. Cependant, il est toujours plus simple de travailler avec les classes spécialisées puisqu'elles offrent des services plus spécialisés. La classe `CClientDC` permet l'affichage dans l'espace client (*Client Area*) d'une fenêtre. L'espace client est l'espace disponible dans une fenêtre excluant le menu et les différentes décorations d'une fenêtre. Cette classe convient à l'affichage normal d'une application. La classe `CWindowDC` est semblable à `CClientDC` mais elle permet

l'affichage dans tout l'espace d'une fenêtre, y compris l'espace réservé pour les décorations (menu, barre d'état, etc.).

Le système Windows indique à une fenêtre de redessiner son contenu par le message `WM_PAINT`. Cette situation se présente lorsqu'une fenêtre est obscurcie par une autre ou lorsqu'elle change de l'état iconique à l'état taille maximale. Ce message est tellement utilisé par Windows qu'une classe dédiée a été implantée par le MFC. La classe `CPaintDC` sert à répondre au message `WM_PAINT` de Windows. Elle est normalement utilisée dans la fonction membre de gestion `OnPaint()` de la classe `CWnd`.

Enfin, la classe `CMetaFileDC` est une classe spéciale dédiée pour traiter le format Metafile de Windows. Le format de fichier Metafile est un ensemble de structure de données pour le stockage des images et des dessins. Le format Metafile est également indépendant de la résolution des périphériques. Il est donc possible d'afficher une image ou un dessin Metafile convenablement sur tous les périphériques supportés par Windows.

Nous allons utiliser un programme simple afin de montrer l'utilisation de ces classes d'affichage. D'abord, créer un programme utilisant un panneau de dialogue comme fenêtre principale. Les étapes pour la création du programme sont énumérées ci-dessous[1] :

Création du projet

1. Démarrer Visual C++ et sélectionner File → New.

2. Dans l'onglet Project, sélectionner MFC AppWizard (exe) et donner le nom `ExempleDC` au paramètre Name. Cliquer sur le bouton OK.

3. Choisir Dialog Based Application dans le panneau de paramètres de l'étape 1.

4. Cliquer sur le bouton Finish. La paramétrisation de l'application est maintenant terminée.

5. Cliquer OK pour fermer la fenêtre de résumé.

---

[1] Cet exemple est tiré du livre "Using Visual C++ 6", Que, 1998.

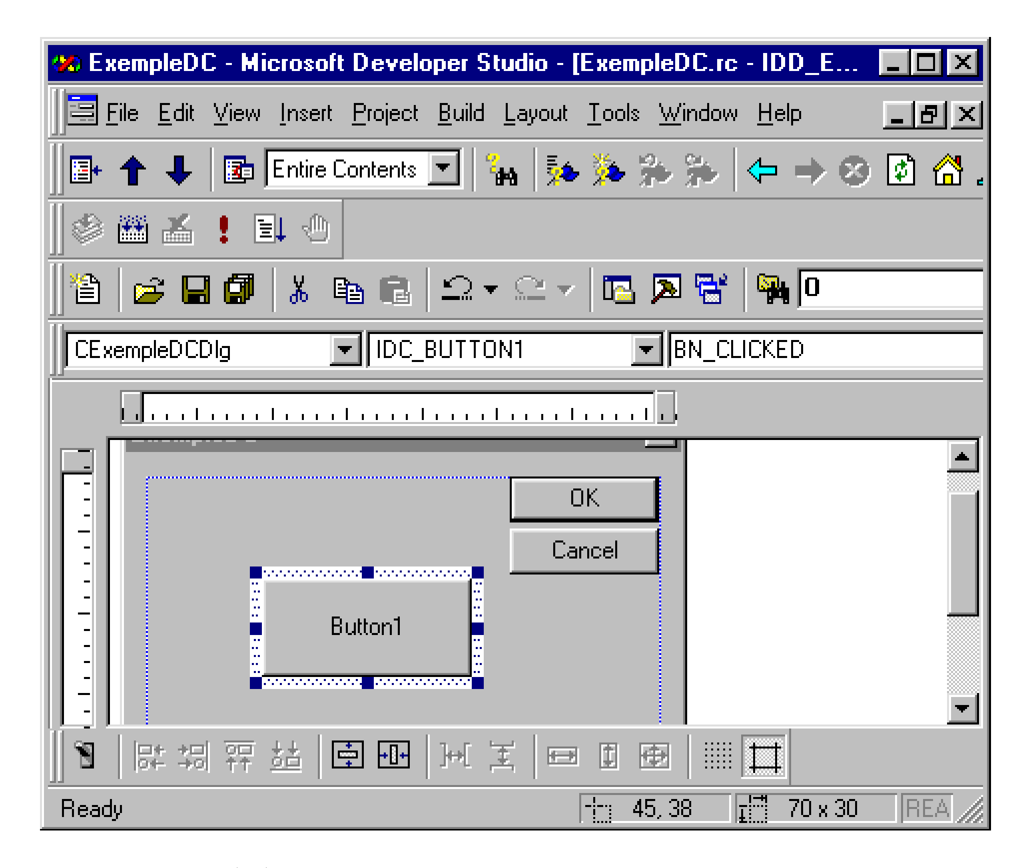


Figure 17 Panneau de dialogue en construction.

Création du bouton

1. Activer l'onglet ResourceView de la fenêtre de Workspace.

2. Ouvrir le dossier Resources de `ExempleDC`.

3. Double-cliquer sur le panneau de dialogue `IDD_EXEMPLEDC_DIALOG` et effacer le texte de rappel ajouté par AppWizard (i.e. AFAIRE: ou TODO:).

4. Placer un bouton dans le panneau de dialogue. Le bouton est disponible dans la barre d'outils flottante de l'éditeur de ressources. Agrandir le bouton pour obtenir une taille convenable (voir Figure 17).

5. Changer l'étiquette de ce bouton en `Dessiner!`. Pour ce faire, sélectionner le bouton, cliquer le bouton droit de la souris et choisir l'option Properties. Profiter l'occasion pour changer l'identificateur `IDC_BUTTON1` en `IDC_EXEMPLEDC`.

Instauration d'un gestionnaire de message

1. Le message Windows à traiter pour le bouton `Dessiner!` est `BN_CLICKED`. Ce message de notification est déclenché à chaque sélection du bouton.

2. Sélectionner le bouton `Dessiner!` et appuyer sur la combinaison `ctrl-W` pour démarrer le ClassWizard.

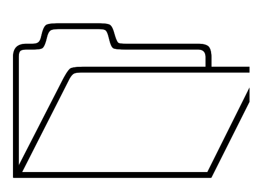
3. Choisir dans Object IDs: l'identificateur IDC_EXEMPLEDC. Choisir le message BN_CLICKED dans Messages: et sélectionner le bouton Add Function. Accepter le nom de la fonction membre par proposé.

4. Cliquer sur Edit Code pour commencer l'édition de la fonction membre.

5. Ajouter le code de traçage ci-dessous dans la fonction membre CExempleDCDlg::OnExempledc(). Noter que le code utilise un objet CDC afin de dessiner sur le bureau de Windows. La fonction de dessin utilisée est SetPixel(), une fonction membre de CDC.

```
void CExempleDCDlg::OnExempledc()
{
 // 1) prendre un pointeur du bureau
 CWnd* pDesktop = GetDesktopWindows();
 // 2) prendre un pointeur du DC
 CDC* pDC = pDesktop->GetWindowDC();
 // 3) créer un dessin sur le bureau !
 for (int x=0; x<300; x++)
  for (int y=0; y<300; y++)
   // Mettre des pixels en couleur
   pDC->SetPixel(x, y, x * y);
 // 4) libérer le DC
 pDesktop->ReleaseDC(pDC);

}
```



À noter que la libération du DC par la fonction ReleaseDC est obligatoire seulement pour la classe CDC. Toutes les classes dérivées de CDC effectuent la libération du DC automatiquement dans leur destructeur. L'exception à cette règle est la création dynamique du DC, dans ce cas, vous devez détruire le DC explicitement par l'instruction delete.

Nettoyage de l'écran

1. Le traçage sur le bureau laisse des traces indésirables. Nous devons nettoyer le bureau à la sortie du programme. Une façon correcte de remettre le bureau en ordre est la capture du message WM_DESTROY de Windows. Lorsque l'utilisateur sélectionne le bouton OK/Annuler du panneau de dialogue[2], le message WM_DESTROY est automatiquement envoyé au programme.

2. Sélectionner l'onglet ClassView de la fenêtre Workspace. Choisir la classe nommée CExmepleDCDlg et cliquer sur le bouton droit de la souris. Sélectionner l'option Add Windows Message Handler pour ajouter un gestionnaire de message pour WM_DESTROY. À noter que la classe CExempleDCDlg est une classe générée par AppWizard lors de la création du projet.

3. Choisir dans la fenêtre New Windows Messages/Events: le message WM_DESTROY puis cliquer sur le bouton Add and Edit. L'IDE de Visual C++

---

[2] WM_DESTROY est également généré après la sélection de l'option Close du menu système ou encore la combinaison alt-F4 du clavier.

afficher dans l'éditeur de texte, la définition de la fonction membre `OnDestroy()`.

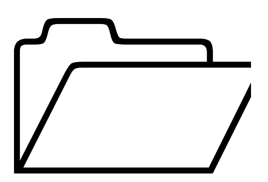
4. Ajouter le code suivant dans l'éditeur de texte.

```
void CExempleDCDlg::OnDestroy()
{
    // Exécuter OnDestroy de la classe de base
    // Cette ligne a été générée par ClassWizard
    CDialog::OnDestroy();

    // 1) Redessinrer le bureau et toutes les fenêtres
    //    du bureau
    GetDesktopWindow()->RedrawWindow(NULL, NULL,
        RDW_ERASE + RDW_INVALIDATE
        + RDW_ALLCHILDREN + RDW_ERASENOW);

}
```

À la réception du message `WM_DESTROY`, le contrôle du programme est passé à la fonction membre `CExempleDCDlg::OnDestroy()`. Cette fonction exécute d'abord `OnDestroy()` de la classe de base (`CDialog`) puis demande à tous les éléments du bureau de redessiner leur interface par la fonction `RedrawWindow()` de l'objet bureau.



Le **chaînage**, c'est-à-dire, l'exécution de la fonction membre correspondante de la sur-classe est très utilisé dans le MFC. Le chaînage est nécessaire parce qu'un message routé à un objet n'est plus disponible aux objets de la surclasse. Dans le cas de la fonction membre `OnDestroy()`, il est nécessaire de permettre à l'objet d'effectuer le nettoyage du bureau mais aussi de se débarrasser des données intrinsèques à un panneau de dialogue.

Ces données sont gérées par la sur-classe `CDialog` et c'est pour cette raison que l'on doit appeler la fonction membre `OnDestroy()` de la sur-classe. Enfin, l'emplacement (i.e. au début ou à la fin de la fonction) du chaînage dans une fonction membre est important. Consulter l'aide en-ligne de Visual C++ pour connaître si le chaînage est nécessaire pour une fonction MFC et son emplacement dans le codage.

Le résultat de l'exécution du programme `ExempleDC` est montré dans la Figure 18. Noter que l'utilisation de la classe `CDC` permet l'affichage n'importe où sur le bureau. Un avertissement est nécessaire ici. L'affichage sur le bureau n'est pas vraiment conseillé. Le risque de corruption est très grand et peut ennuyer les utilisateurs du programme.

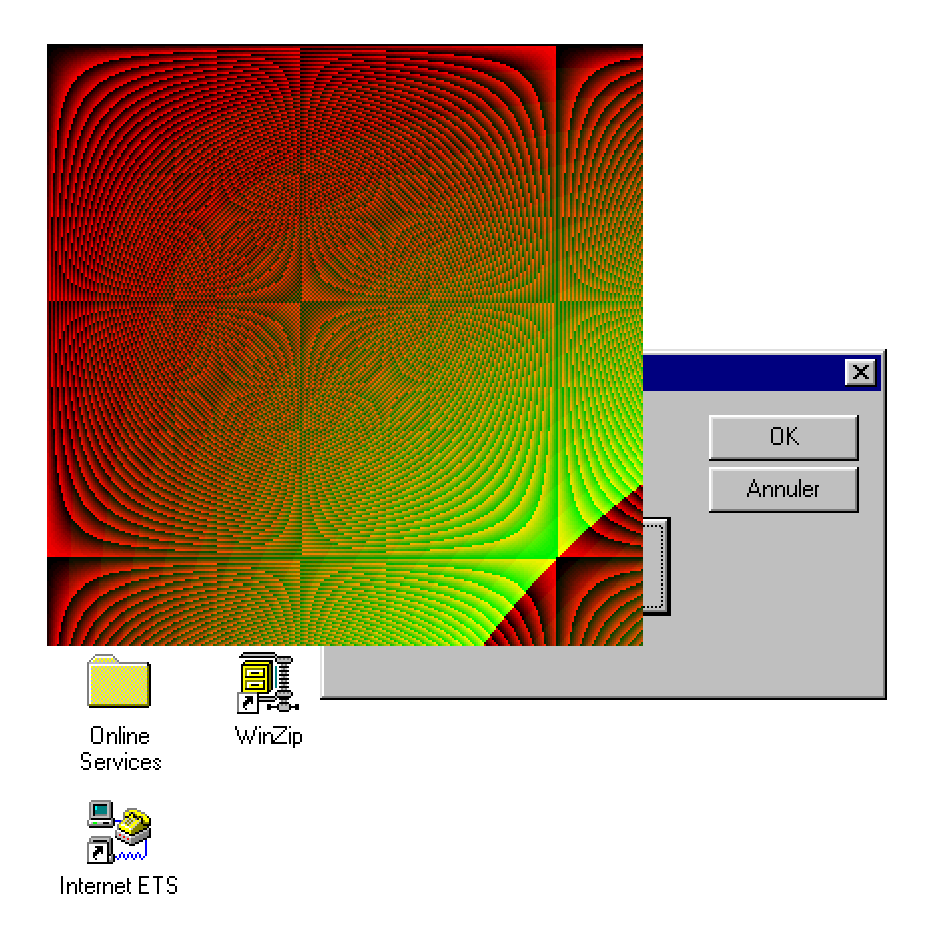


Figure 18 Avec CDC, il est possible de dessiner n'importe où sur le bureau.

Affichage dans l'espace client

1. La classe `CClientDC` est l'objet de cet exemple. Avec `CClientDC`, le dessin s'affichagera dans l'espace réservé de la fenêtre.

2. Modifier la fonction membre `CExempleDCDlg::OnExempledc()` et remplacer le `CDC` du bureau par le `CClientDC` du panneau de dialogue.

3. Entrer le code suivant dans la fonction `OnExempledc()`.

La classe `CExempleDCDlg` est dérivée de la classe `CDialog`, elle-même dérivée de `CWnd`. Un objet `CClientDC` est obtenu en utilisant le pointeur du panneau de dialogue (fenêtre) comme paramètre. Le traçage s'effectuera dans l'espace réservé du panneau de dialogue. Ce résultat est montré dans la Figure 19.

```
void CExempleDCDlg::OnExempledc()
{

 // 1) construire un client DC du panneau
 //    de dialogue (CDialog est dérivé
 //    de CWnd)
 CClientDC DlgDC(this);

 // 2) créer un dessin sur le bureau !
 for (int x=0; x<150; x++)
```

```
for (int y=0; y<150; y++)
 // Mettre des pixels en couleur
 DlgDC.SetPixel(x, y, x * y);
// 3) libérer le DC
// Pas besoin pour les classes dérivées
// de CDC !

}
```

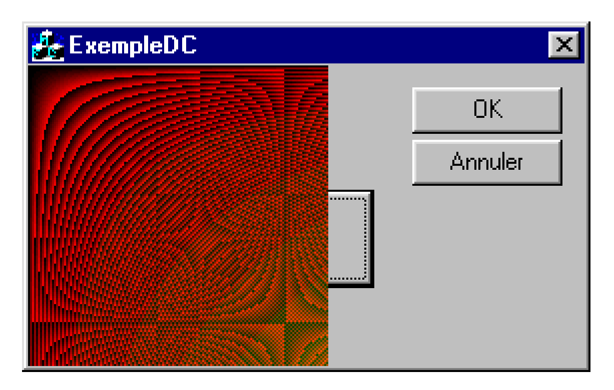


Figure 19 Résultat de l'utilisation de CClientDC.

3.3.4 SYSTÈMES DE COORDONNÉES

Le système Windows possède plusieurs modes d'affichage et utilise deux types de coordonnées : *i*) coordonnées logiques; *ii*) coordonnées physiques. Une unité logique peut correspondre à une ou plusieurs unités physiques. Par exemple, les coordonnées physiques de l'écran sont toujours exprimées en pixel. Tandis que les coordonnées logiques peuvent être exprimées en pouces ou en cm.

Le mode d'affichage par défaut dans Windows est le `MM_TEXT`. Dans le mode `MM_TEXT`, les coordonnées logiques et physiques sont les mêmes. Ainsi, une unité logique correspond exactement à une unité physique. Ce mode est ainsi nommé parce que, semblable à du texte, l'origine des coordonnées (0, 0) est située dans le coin supérieur gauche. Les axes *x* et *y* progressent positivement vers la gauche et vers le bas. Le tableau II donne les différents mode d'affichage possible.

MODE	Une unité logique est équivalente à
MM_TEXT	Un pixel
MM_LOMETRIC	0.1 millimètre
MM_HIMETRIC	0.01 millimètre
MM_LOENGLISH	0.01 pouce
MM_HIENGLISH	0.001 pouce
MM_TWIPS	1/1440 d'un pouce
MM_ISOTROPIC	Valeur définissable par programmation. L'axe x et y sont identiques
MM_ANISOTROPIC	Valeur définissable par programmation. L'axe x et y sont indépendants

Figure 20 Modes d'affichage possibles.

Le mode d'affichage et les systèmes de coordonnées font partie des attributs d'un DC. La fonction membre `SetMapMode()` de la classe de base `CDC` permet au programmeur de changer aisément le mode d'affichage. À noter qu'il est tout à fait possible de changer le mode d'affichage en cours d'exécution du programme. Ainsi, une unité logique correspond à 0.01 pouce peu importe la résolution des périphériques utilisés.

Les modes `MM_ISOTROPIC` et `MM_ANISOTROPIC` sont utiles pour représenter des unités logiques particulières qui ne figurent pas dans le tableau II. Par exemple, on peut représenter une unité logique comme 1/72 pouce en unité physique. Dans le cas de `MM_ISOTROPIC`, les axes *x* et *y* ont la même correspondance entre une unité logique et une unité physique. On peut obtenir des effets intéressants à l'aide du mode `MM_ANISOTROPIC` puisque ce dernier peut avoir des correspondances différentes pour les axes. En utilisant ce mode, une unité logique en *x* peut ne pas avoir la même longueur que celle en *y*. Enfin, à l'aide du mode `MM_ANISOTROPIC` on peut réaliser facilement l'option *Zoom*.

3.3.5 CRAYONS ET PINCEAUX

Les instruments de traçage disponibles sont les crayons et les pinceaux. Les crayons servent à dessiner des traits alors que les pinceaux sont utiles pour peindre l'intérieur d'une figure fermée.

La classe `CPen` et la classe `CBrush` encapsulent les structures de données nécessaires pour la manipulation de ces instruments de traçage. Par défaut, un DC contient un crayon de couleur noire, d'un style de type *ligne pleine* et d'une épaisseur d'une unité logique. L'épaisseur est donc une fonction du mode d'affichage en vigueur dans le DC au moment du traçage. Par défaut, un DC contient un pinceau de couleur transparente (?) et d'un style de type plein.

Il existe deux sortes de crayons et de pinceaux dans le système Windows. Les crayons et pinceaux prédéfinis sont disponibles par l'utilisation de la fonction membre `SelectStockObject()` de la classe `CDC` (et ses dérivées). L'autre sorte est construite à partir des constructeurs de `CPen` et `CBrush`. Ces ressources de traçage sont ensuite placées dans le DC à l'aide de la méthode `SelectObject()` de la classe `CDC` (et ses dérivées).

Donc, il est toujours nécessaire de : *i*) placer les instruments dans le DC; *ii*) effectuer le traçage en utilisant les fonctions membres de `CDC` (`CClientDC`, `CPaintDC`, etc.); *iii*) enlever les instruments du DC. De plus, si les instruments sont créés dynamiquement (par l'instruction `new`) alors on doit les détruire (par `delete`) à la fin de la l'étape *iii*. Ceci est absolument nécessaire parce que Windows ne dispose qu'un nombre limité de ressources représentant les instruments de traçage[3]. Sans quoi, une erreur fatale peut survenir à cause de cette perte de ressources.

---

[3] En fait, il s'agit aussi une limitation dans le nombre de DC disponible.

3.3.6 AUTRES INFORMATIONS

Les classes `CDC`, `CClientDC`, `CPaintDC` et `CMetaFileDC` contiennent un grand nombre de services. Notamment, les fonctions de traçage (`LineTo()`, `MoveTo()`, `DrawText()`, etc.), les fonctions de conversion de l'unité logique en unité physique et vice versa (`LPtoDP()`, `DPtoLP()`), les fonctions qui modifient l'origine des axes (`SetViewPortOrg()`, `SetWindowOrg()`) et les fonctions qui modifient l'étendue de la surface de traçage (`SetViewportExt()`, `SetWindowExt()`). Il est possible de connaître les détails de ces fonctions membres à l'aide de l'aide en-ligne de Visual C++ sous le rubrique MFC.

3.4 ÉLÉMENTS D'UNE APPLICATION MFC

Une application Windows est composée d'un ensemble d'éléments d'interface. De nos jours, les applications Windows ont toutes une certaine présentation minimale qui est de facto. Par exemple, une application résidente est normalement démarrée sous forme iconique et rangée dans la zone de notification de la barre de tâches de Windows (ex : Iomega 1-Step Backup, Norton Program Scheduler, etc.). Un programme de gestion de paramètres ne contient qu'un panneau de paramètres en utilisant des onglets (*Tab controls*) pour diviser les différents paramètres en groupes. Enfin, une application complète doit au minimum posséder un menu, une barre d'outils flottante, un espace de travail sous forme de fenêtre et une barre d'état.

Tous ces éléments d'interface sont encapsulés dans les nombreuses classes du cadre de travail MFC. Ces classes du MFC offrent tous les services nécessaires pour la gestion et la manipulation des éléments d'interface. Ces classes du MFC ne sont pas les représentants graphiques des éléments d'interface. Par contre, elles possèdent comme attributs les structures de données nécessaires pour faire afficher leur représentation à l'écran. Il est très important de distinguer les classes MFC de la représentation graphique des éléments d'interface. Ainsi, il est possible de détruire la représentation graphique d'un élément d'interface sans détruire l'objet MFC associé. De même, il est possible de créer l'objet MFC sans construire la représentation graphique associée. Enfin, pour certains éléments d'interface, la création d'un objet MFC entraîne automatiquement l'affichage de la représentation graphique. Tandis que, pour d'autres éléments d'interface, on doit effectuer explicitement l'affichage de la représentation graphique à l'aide de fonctions membres appropriées.

3.4.1 MENU ET BARRE D'OUTILS

La création de la représentation graphique d'un menu passe par l'éditeur de ressources. Normalement, l'outil AppWizard aurait déjà généré cette ressource à partir des informations données lors de la création du projet. Consulter l'aide en-ligne pour connaître les détails nécessaires pour la manipulation des menus dans l'éditeur de ressource.

Pour la barre d'outils, encore une fois l'outil AppWizard doit avoir déjà généré cette ressource. La création de nouveaux boutons est aussi simple que l'ajout d'une

nouvelle option dans un menu. Cependant, vous devez exercer votre talent d'artiste pour la création des images à l'intérieur des boutons.

La création de l'objet MFC du menu et son insertion dans le cadre principal de l'application sont réalisées dans la fonction `CGPA789App::InitInstance()`. Une fois initialisé, le menu devient autonome.

```
// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
 return FALSE;
m_pMainWnd = pMainFrame;
```

Le code ci-dessus est généré par l'outil AppWizard. Lorsqu'une option est sélectionnée par l'utilisateur, le menu envoie un message de commande à l'application avec comme paramètre, l'identificateur représentation l'option sélectionnée. C'est pour cette raison que la carte de message possède le macro `ON_COMMAND()`. Ce dernier sert justement à capturer les messages de commande des menus et des barres d'outils.

Pour l'objet MFC de la barre d'outils, il est créé et inséré dans le cadre principal de l'application lors de la création du cadre par l'instruction : `CMainFrame* pMainFrame = new CMainFrame;` La mécanique de sa création est explicitée dans la fonction `CMainFrame::OnCreate()` contenue dans le fichier source `MainFrm.cpp`. Ce dernier est également généré par AppWizard. La barre d'outils produit les mêmes messages de commande que le menu.

3.4.2 MENUS SURGISSANTS

Les menus surgissants (*Popup Menu*) sont utiles pour donner un menu contextuel qui facilite l'utilisation du logiciel. Un menu surgissant est normalement créé lorsque l'utilisateur appuie sur le bouton droit de la souris. Son contenu varie selon certaines conditions déterminées. Par exemple, l'activation du bouton droit de la souris affichera un menu surgissant montrant les outils de dessin disponibles. Par contre, si la souris est au-dessus d'un élément de dessin alors, le menu surgissant présentera les options d'édition.

Pour créer un menu surgissant, il faut d'abord créer sa représentation graphique à l'aide de l'éditeur de ressources. De plus, il est important d'assigner les bons identificateurs aux options.

Il s'agit de la responsabilité d'une vue de montrer les menus surgissants. Après tout, c'est une fenêtre associée à une vue qui reçoit le message du bouton droit activé. Donc, le code suivant peut être utilisé.

```
void CGPA789View::OnRButtonDown(UINT nFlags, CPoint
↳ point)
{
 CMenu Menu;
// Charger le menu flottant des ressources
 Menu.LoadMenu(IDR_MENU_CURSEUR);
```

```
// Convertir en coord. écran
  ClientToScreen(&point);
  Menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
➥ point.x, point.y, this);
}

```

Dans ce fragment de code, le menu surgissant `IDR_MENU_CURSEUR` s'affichera dès que le bouton droit de la souris est appuyé par l'utilisateur.

3.4.3 BARRES D'ÉTAT

Une barre d'état est présente dans le cadre principal et dans les fenêtres enfants d'une application MDI. La barre d'état du cadre principal est créée dans la fonction membre `CMainFrame::OnCreate()` en même temps que la barre d'outils. Quant aux fenêtres enfants, la barre d'état est toujours associée au cadre de la fenêtre et non à la vue. La raison est que la vue peut disposer d'une barre de défilement, et si la barre est associée à la vue alors il devient possible de défiler la barre d'état hors de la région visible de la fenêtre !

Ainsi, la barre d'état des fenêtres enfants est créée dans un objet de la classe `CChildFrame` en utilisant la fonction membre `OnCreate()`. La classe `CChildFrame` représente le cadre d'une fenêtre enfant. Rappelons qu'un cadre de fenêtre est responsable de la gestion de la barre de titre, de la bordure et dans ce cas, la barre d'état. Mais nous verrons plus tard que l'importance du rôle d'un cadre est très réduite par le concept DOC/VIEW. Néanmoins, si nous voulons ajouter une barre d'état, c'est le cadre qui demeure responsable.

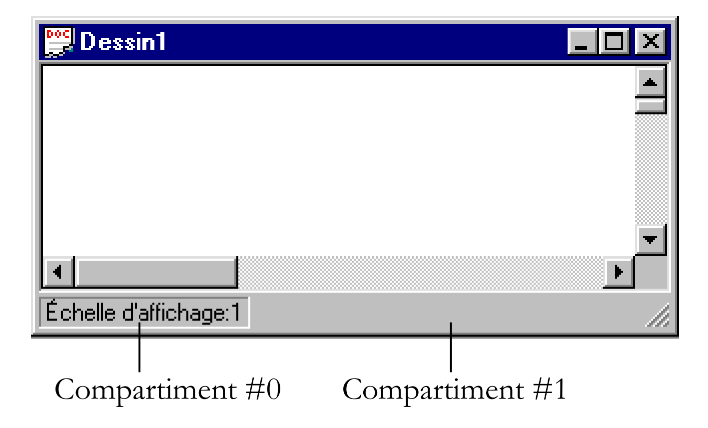


Figure 21 Exemple d'une barre d'état.

Une barre d'état est divisée en compartiments. La Figure 21 est un exemple d'une barre d'état à deux (2) compartiments. Dans cette figure, le premier compartiment porte un encadrement de type creusé. Le style des compartiments est programmable. Le code ci-dessous réalise la barre d'état de la Figure 21. La variable d'instance `m_BarreEtat` est un attribut de la classe `CChildFrame` (ajouté par le programmeur) de type `CStatusBar`. Ici, nous avons un exemple de la séparation entre un objet MFC et la représentation graphique de l'objet. La variable `m_BarreEtat` est un objet

de la classe `CStatusBar`. Cet objet `m_BarreEtat` est construit dès que l'on crée une instance de `CChildFrame`. Or, la représentation graphique de la barre d'état n'est créée que plus tard dans la fonction membre `OnCreate()` de `CChildFrame`.

```
int CChildFrame::OnCreate(LPCREATESTRUCT
↳ lpCreateStruct)
{
 if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1)
  return -1;

 // 1) Créer la barre d'état
 m_BarreEtat.Create(this);

 // 2) Ajuster la grandeur des cases en fonction de la
↳ taille des
 //    caractères à afficher
 CRect Rect;
// Une barre d'état est aussi une fenêtre !
 CClientDC dc(&m_BarreEtat);        dc.SelectObject(m_BarreEtat.GetFont());
 dc.DrawText("ÉChelle d'affichage:99", -1, Rect,
↳ DT_SINGLELINE | DT_CALCRECT);

 int Largeur = Rect.Width();        // Ajuste la grandeur de la première case
 m_BarreEtat.GetStatusBarCtrl().SetParts(1, &Largeur);

 // 3) Initialiser le texte
 m_BarreEtat.GetStatusBarCtrl().SetText("Échelle
↳ d'affichage:1", 0, 0);

 return 0;
}
```

Plus subtile encore, la classe `CStatusBar` contient en réalité une autre classe appelée `CStatusBarCtrl`. C'est cette dernière qui interagit avec la représentation graphique de la barre d'état. Ainsi, on doit obtenir un pointeur de `CStatusBarCtrl` (par la fonction membre `GetStatusBarCtrl()`) et utiliser ses services pour obtenir les effets désirés.

3.4.4 PANNEAUX DE DIALOGUE

L'une des techniques pour rendre un logiciel interactif est l'utilisation des panneaux de dialogue. Les panneaux de dialogue servent à obtenir de l'utilisateur des paramètres ou une confirmation de l'action à entreprendre. Prenons l'exmple d'un panneau de dialogue est prévu pour obtenir l'épaisseur du crayon. Il est affiché en réponse à la sélection de l'option Épaisseur du menu Crayon (ou de la barre d'outils et du menu surgissant). La Figure 22 donne un exemple du panneau de dialogue utilisé.

À noter que la représentation exacte de ce panneau de dialogue peut varier. Le but ici est d'obtenir une valeur numérique qui représente l'épaisseur du crayon de dessin. Ainsi, nous pouvons utiliser les boutons de radio (tels que montrés dans la figure 14) ou une liste de sélection donnant toutes les valeurs permises dans une liste déroulante. Dans le langage de Windows, tous ces éléments sont appelés des éléments de contrôle.



Figure 22 Exemple d'un panneau de dialogue.

Dans le MFC, les panneaux de dialogue sont des éléments distincts de l'application. Le programmeur doit produire leurs propres panneaux de dialogue[4]. Tout comme les autres éléments d'interface de Windows, la création d'un panneau de dialogue s'effectue en deux étapes. D'abord, il faut créer les éléments graphiques du panneau de dialogue à l'aide de l'éditeur de ressources. Ensuite, nous ajoutons le codage nécessaire pour rendre le panneau de dialogue utile dans l'application.

Deux types de dialogue existent dans Windows. Le type modal affiche le panneau de dialogue puis suspend l'opération de l'application et passe le contrôle au panneau de dialogue. Le contrôle est retourné à l'application après la fermeture du panneau de dialogue. Le type non modal n'assume pas le contrôle de l'application et peut demeurer actif en même temps. Le panneau de paramètres pour la sélection de l'épaisseur du crayon de dessin est du type modal.

Pour rendre un panneau de dialogue utile, il faut créer pour lui une nouvelle classe dans l'application. En utilisant l'outil ClassWizard, on peut créer une classe `CCrayonDlg` et assigner l'identificateur du panneau de dialogue dans les paramètres exigés. La classe `CCrayonDlg` sera dérivée de `CDialog`, la classe de base des panneaux de dialogue de MFC.

Le fonctionnement d'un panneau de dialogue est régi par les éléments de contrôle associés. Par exemple, dans le panneau de la figure 13, il existe six (6) éléments de contrôle. Chaque contrôle (dans ce cas un bouton) doit avoir un identificateur différent. On peut procéder selon le schème où le bouton étiqueté "1 pixel" possède l'identificateur `IDC_EPAISSEUR0`, le bouton étiqueté "0.01 pouce" possède l'identificateur `IDC_EPAISSEUR1`, et ainsi de suite. Lorsque l'utilisateur sélectionne l'un des six boutons, un message de notification est généré par le bouton et le message est envoyé au panneau de dialogue. Remarquer qu'il y a découplage total entre le panneau de dialogue et l'application. Les messages de notification des

[4] Évidemment, si le programme utilise un panneau de dialogue comme fenêtre principal, l'outil AppWizard peut générer le code source initial pour la création et la gestion du panneau de dialogue.

éléments de contrôle sont dirigés vers le panneau de dialogue et non vers l'application[5].

Il est donc nécessaire de réaliser les gestionnaires de message directement dans la classe `CCrayonDlg`. Le message généré par les contrôles est `BN_CLICKED` avec comme paramètre l'identificateur du bouton sélectionné. À noter que le comportement *radio* (c'est-à-dire, mutuellement exclusif) des boutons est réalisé automatiquement par le MFC avec l'aide de l'éditeur de ressources.

Quant à l'application, sa responsabilité consiste à faire afficher le panneau de dialogue, obtenir la valeur représentant l'épaisseur du crayon à la fermeture du panneau. L'endroit idéal pour réaliser ces tâches est dans le document de l'application (voir Concept DOC/VIEW à la page 125). Ainsi, après la sélection de l'option Épaisseur du menu Crayon, le message de commande `ID_EPAISSEUR` est routé à la fonction membre déterminée par la carte de message du document.

```
void CGPA789Doc::OnEpaisseur()
{
 CCrayonDialog Dlg;

 // 1) Donner l'épaisseur du crayon à l'objet Dlg
 Dlg.m_Epaisseur = m_EpaisseurCrayon;
 // 2) Afficher le panneau de dialogue
 if (Dlg.DoModal() == IDOK)
  m_EpaisseurCrayon = Dlg.m_Epaisseur;

}
```

Dans cet exemple, la fonction membre `OnEpaisseur()` de la classe document `CGAP789Doc` est le gestionnaire du message. D'après le codage, il est clair que le panneau de dialogue `Dlg` contient une variable d'instance `m_Epaisseur` qui est publique. L'exécution du panneau de dialogue est réalisée par la fonciton `DoModal()` de l'objet `Dlg`. Le contrôle est alors passé à l'objet `Dlg` et lorsque le panneau termine son exécution, on vérifie l'état de sa fermeture. La fonction `DoModal()` retourne l'identificateur `IDOK` si la fermeture du panneau de dialogue est réalisée par l'activation du bouton OK. Si tel est le cas, on prend le contenu de `m_Epaisseur` et l'emmagasine dans le document.

Le mécanisme d'échange entre le panneau de dialogue et l'application montrée ici utilise une variable d'instance publique. Il s'agit du mécanisme le plus simple mais pas nécessairement le plus efficace. Mais il suffit amplement pour nos besoins.

Il existe cependant un autre mécanisme de gestion de données qui est fort utile. Cette fois, il concerne les échanges de données entre les contrôles et le panneau de dialogue. Le DDX (*Dialog Data eXchange*) permet d'automatiser l'échange des paramètres reçus des contrôles et le panneau de dialogue. Ce mécanisme existe pour les contrôles qui reçoivent des valeurs de la part de l'utilisateur. Par exemple, le panneau de dialogue pour la modification de l'échelle d'affichage (*Zoom*) exige

[5] Bien sûr, il est possible de notifier l'application directement mais nous n'entrons pas dans ces détails. Consulter l'outil InfoView pour connaître le mécanisme d'échange entre un panneau de dialogue et une application mère.

l'entrée des valeurs numériques. Pour transférer et valider automatiquement la valeur de l'échelle, on peut recourir à la méthode DDX (et sa consœur DDV pour validation). L'outil ClassWizard est responsable d'ajouter automatiquement le mécanisme de DDX (et DDV) dans le panneau de dialogue. Donc, il est conseillé de créer les classes de contrôles et les classes de panneaux de dialogue à l'aide de ClassWizard.

3.4.5 SÉRIALISATION DES OBJETS

Un document dans le concept DOC/VIEW peut contenir une myriade d'objets. L'enregistrement d'un document sur disque n'est pas trivial surtout pour des applications de grande complexité. Si la vision choisie est centrée sur les données, alors nous pouvons enregistrer uniquement les données de l'application sur disque. Le problème est malheureusement apparent lors de la lecture du fichier de données. Pour pouvoir recréer les objets, nous devons disposer d'informations supplémentaires (au minimum l'identification de l'objet à créer). Si la structure des objets est complexe (ex. : Objets imbriqués) alors les informations supplémentaires seront aussi difficiles à gérer.

Dans la plupart des cas, l'enregistrement des données de l'application ne suffit pas à la tâche. C'est ainsi que la sérialisation des objets peut aider grandement. De plus, la sérialisation permet le transfert des objets d'une application vers une autre. Cependant, nous n'insisterons que sur l'aspect enregistrement et chargement de la sérialisation.

L'un des critères de la sérialisation est l'existence d'un lien de parenté entre une classe à sérialiser et la classe abstraite de base `CObjet`. En effet, une classe qui désire réaliser la sérialisation doit être dérivée de la classe `CObjet` et surcharger la fonction membre virtuelle `Serialize()`. Dans cette fonction virtuelle, un objet (instance d'une classe) doit s'intéresser uniquement de l'enregistrement (ou chargement) des données qui le concerne. L'objet n'a pas à s'occuper de son numéro d'identification ou encore de l'ordre de sa sauvegarde (ou chargement).

Dans la sérialisation, tout le travail impliqué dans la création des objets et dans l'ordre de la création est réalisé par le MFC. Les informations supplémentaires pour recréer les objets lors de la lecture du fichier de données sont enregistrées (lors de la sauvegarde) par un objet utilitaire MFC, une instance de la classe `CArchive`. Le MFC crée automatiquement un objet `CArchive` à la réception des messages de sauvegarde ou chargement (Enregistrer, Enregistrer Sous, Ouvrir). Le MFC exécute ensuite, dans un ordre convenable, la fonction `Serialize(CArchive& ar)` de tous les objets sérialisés de l'application avec comme paramètre un objet `CArchive`. Le MFC connaît l'existence de tous les objets sérialisés de l'application puisqu'ils sont tous dérivés de `CObjet`.

L'objet `CArchive` encapsule en lui, un pointeur du fichier obtenu de MFC[6] et la logique nécessaire pour créer un fichier de données sérialisées. En pratique, `CArchive` enregistre sur disque *i*) les données de l'objet (responsabilité de l'objet); *ii*) le type de l'objet (responsabilité de MFC et de `CArchive`); *iii*) la version de l'objet (responsabilité du programmeur, MFC et `CArchive`). Le code ci-dessous est un exemple de sérialisation d'un objet appelé `CLigne`.

```
void CLigne::Serialize(CArchive& ar)
{
 // 1) Appel de la classe de base
 CElement::Serialize(ar);

 // 2) Notre tour ...
 if (ar.IsStoring()) {
  ar << m_PointDepart
     << m_PointFin;
 }
 else {
  ar >> m_PointDepart
     >> m_PointFin;
 }
}
```

Remarquer que `CLigne` exécute toujours en premier la sérialisation de son parent (`CElement`). Cette façon de faire permet à la sérialisation de parcourir toute la hiérarchie des classes. L'objet `ar` de la classe `CArchive` peut nous indiquer l'état de la sérialisation (`ar.IsStoring()`). Le point le plus important ici que l'on doit écrire ou lire à travers l'objet `ar` et non directement sur un flux d'E/S.

On a vu plus haut, le rôle de l'objet sérialisé. Quelle est la responsabilité du programmeur dans la sérialisation? Dans le contexte de la sérialisation, le programmeur doit indiquer dans le code source, les classes qui seront sérialisés par l'insertion des macros Visual C++ (MFC) `DECLARE_DYNACREATE()` et `IMPLEMENT_DYNACREATE()`dans le document de l'application. Les macros `DECLARE_DYNACREATE()` et `IMPLEMENT_DYNACREATE()` permettent la création dynamique des classes de l'application par le MFC lors de la phase chargement de la sérialisation (*serialisation input*). Le programmeur doit également s'assurer que `CObjet` figure parmi les ancêtres des classes sérialisées.

De plus, le programmeur doit signaler qu'une classe est sérialisée par l'emploi des macros `DECLARE_SERIAL()` et `IMPLEMENT_SERIAL()`. Il est nécessaire de placer ces macros dans toutes les classes sérialisées de l'application. Enfin, c'est dans le macro `IMPLEMENT_SERIAL()` que l'on doit désigner un numéro de version des objets sérialisés. Par exemple, pour les objets de `CLigne`, la version peut être 1 tandis que les objets de `CTexte` peut avoir un numéro de version 6. Le numéro de version des objets simplifie la gestion de configuration du logiciel. Consulter la document en-ligne pour connaître la syntaxe et l'utilisation de ces macros.

[6] Le nom du fichier est en réalité obtenu de l'utilisateur par les panneaux de dialogue associés aux options Enregistrer, Enregistrer Sous et Ouvrir du menu Fichier.

3.4.6 SOUS-SYSTÈME D'IMPRESSION

Le AppWizard est en mesure de générer la capacité d'impression pour une application. Cette capacité comprend les options Fichier → Imprimer, Fichier → Aperçu avant impression et Fichier → Configuration de l'impression. Toutes ces options sont disponibles et actives. Par contre, il est recommandé d'effectuer des ajustements afin de rendre ces options compatibles avec l'application. Par exemple, l'impression d'un dessin peut s'étendre sur plus d'une feuille de 8½ × 11 pouces. Pour pouvoir modifier le résultat des impressions, nous devons savoir comment apporter les changements requis.

3.4.7 PROCESSUS D'IMPRESSION

L'impression d'un document est une tâche complexe mais fondamentalement essentielle. Dans le MFC, le processus d'impression est contrôlé par une vue. La Figure 23 est un schéma résumant ce processus complexe.

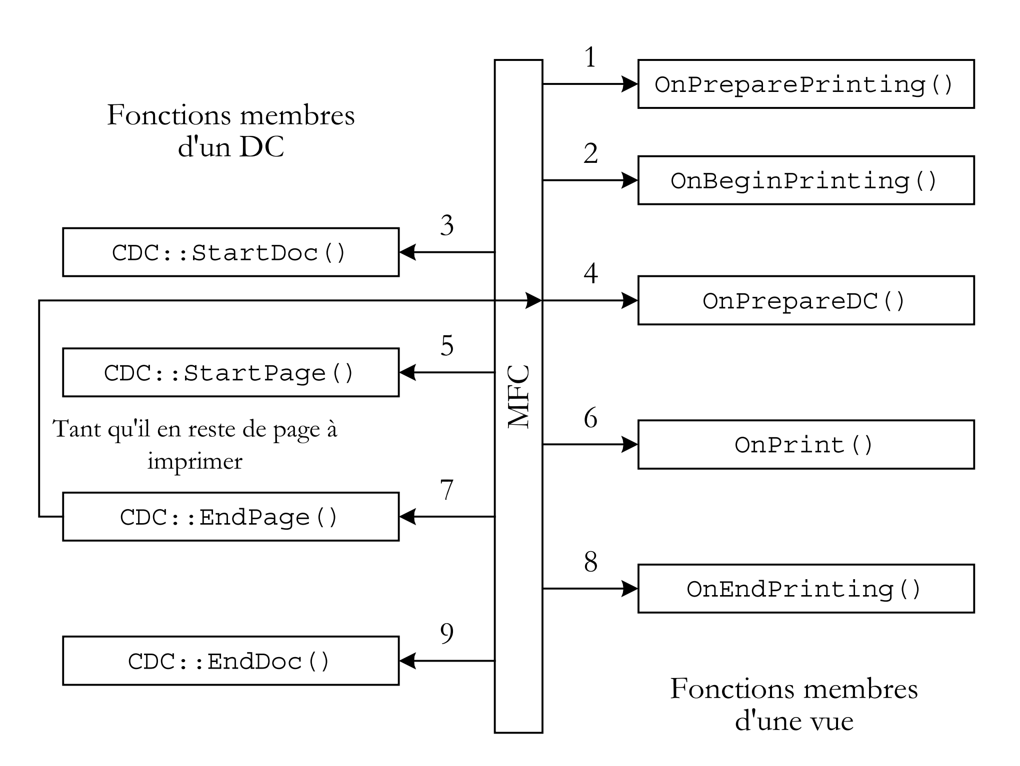


Figure 23 Étapes impliquées dans l'impression d'un document.

L'exécution de la séquence est initiée par le MFC. La fonction `OnPreparePrinting()` est utilisée pour calculer le nombre de page à imprimer. Cette fonction appelle également la méthode `DoPreparePrinting()`. C'est cette dernière qui est responsable d'afficher un panneau de dialogue pour la sélection des paramètres de l'imprimante et la création d'un DC (*Device Contexte*) dédié pour l'impression.

Normalement, nous devons surcharger les fonctions membres `OnPreparePrinting()` (étape 1), `OnPrepareDC()` (étape 4) et `OnPrint()` (étape 6). On voit par la figure 15 que le rôle d'un DC s'étend également sur le processus de l'impression. En effet, l'action d'imprimer implique les mêmes fonctions de traçage que sur l'écran. La différence est dans l'objet `CDC` créé et les fonctions membres utilisées pour la préparation du DC.

Un objet de la classe `CPrintInfo` est utilisé comme paramètre dans toutes les fonctions membres d'une vue impliquées dans le processus d'impression. C'est une façon d'éviter l'utilisation des variables globales dans le schème de la figure 15. Cette classe contient toutes les informations pertinentes à l'impression d'un document. Il est donc important d'examiner son contenu à l'aide de l'outil InfoView.

Nous avons indiqué que la fonction `OnPreparePrinting()` est une fonction membre d'une vue. L'action visible de cette fonction est d'afficher un panneau de dialogue montrant les paramètres de l'impression.

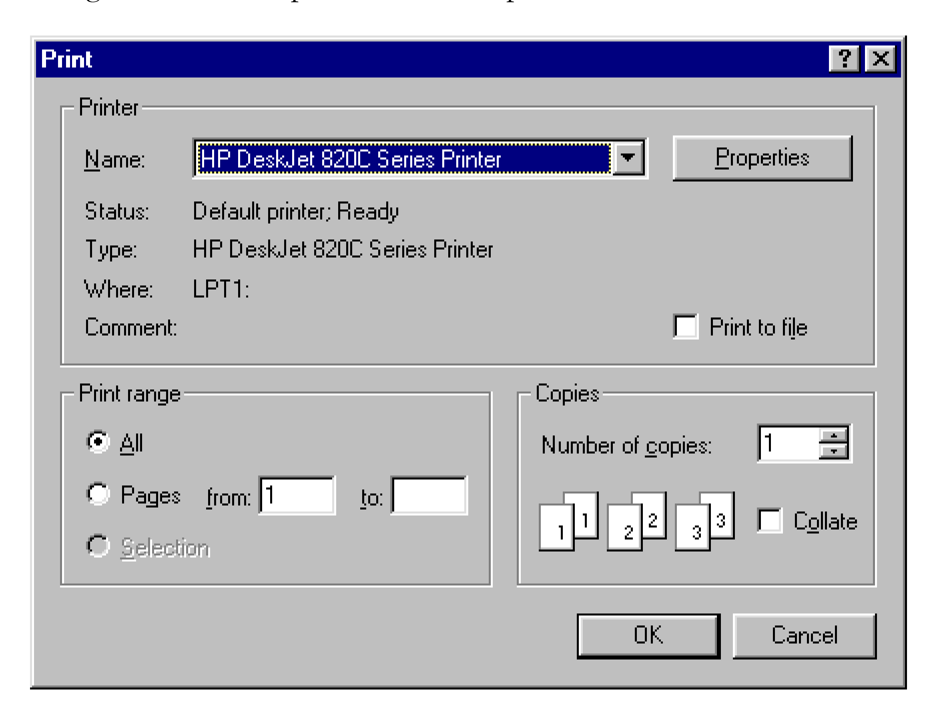


Figure 24 Paramètres initiaux sont obtenus de l'objet CPrintInfo.

Les paramètres initiaux de ce panneau sont ceux contenus dans l'objet `CPrintInfo`. Il est donc important de calculer le nombre de pages du document à imprimer dans la fonction `OnPreparePrinting()` et placer ce paramètre dans l'objet `CPrintInfo`. Après quoi, on peut exécuter la fonction `DoPreparePrintg()` qui produit le panneau de dialogue. Le pseudo-code ci-dessous montre les étapes logiques pour réaliser la fonction `OnPreparePrinting()`.

```
BOOL CGPA789View::OnPreparePrinting(CPrintInfo* pinfo)
{
 // 1) Calculer le nombre de pages à imprimer
 //    L'algorithme de déterminer à réaliser
 //     ci-dessous
```

```
// 2) Placer tous les paramètres obtenus dans
// l'objet CPrintInfo
pInfo→SetMinPage(1); pInfo→SetMaxPage…

// 3) Afficher le panneau de dialogue pour
//    donner à l'utilisateur la possibilité
//    de choisir les paramètres d'impression
if (!DoPreparePrinting(pInfo))
{
 // routine de clean-up
 return FALSE;
}
return TRUE;
}
```

Parce que l'impression d'un document utilise également un DC, nous devons probablement surcharger la fonction `OnPrepareDC()` (si ce n'est pas déjà fait pour les autres besoins de l'application). La raison est la suivante : lors de la création d'un DC, le cadre de travail MFC exécute toujours la fonction `OnPrepareDC()` de la vue active afin de permettre la modification du DC avant qu'il soit utilisé. On peut donc modifier le mode d'affichage (*Mapping Mode*), l'étendue de l'espace d'affichage (*Window Extent*, *Viewport Extent*), etc. directement en un seul endroit (c'est-à-dire dans `OnPrepareDC()`). La fonction membre `OnPrepareDC()` est certainement utilisé pour modifier l'échelle d'affichage d'un dessin. Il est nécessaire, lors de l'impression, de ramener l'échelle à l'unité sans quoi le résultat sera plus que désagréable.

```
void CGPA789View::OnPrepareDC(CDC *PDC,
↳ CPrintInfo* pInfo)
{
 // 1) Remettre l'échelle d'affichage à l'unité
 //    si en mode d'impression
 int Echelle = m_Echelle;
 if (pDC→IsPrinting())
  Echelle = 1;
 // Important! Exécuter OnPrepareDC du parent
 CScrollView::OnPrepareDC(pDC, pInfo);

 // 2) Continuer l'ajustement du DC pour les besoins
 //    du logiciel …
}
```

La fonction membre d'une vue qui réalise l'impression du document est nommée `OnPrint()`. Elle ne figure pas dans le code source initial et nous devons l'ajouter manuellement ou par le ClassWizard. Dans `OnPrint()`, nous calculons la partie du document à imprimer en fonction des paramètres de l'objet `CPrintInfo`. Pour le logiciel de dessin, cela implique la manipulation de l'origine des axes, le calcul de l'étendue d'une page par rapport au dessin et l'exécution de la tracée par les objets (i.e. `CLigne`, `CCercle`, etc.) du dessin.

Enfin, la surcharge de la méthode `OnEndPrinting()` est nécessaire pour éliminer les objets de support créés dynamiquement par le programmeur lors du processus d'impression.

LECTURE SUGGÉRÉE

Les références qui ont aidé à la rédaction de ce chapitre sont :

[BATE98] Bates, J., Tompkins, T., *Using Visual C++ 6*, Que Corporation, 1998.

[KRUG98] Kruglinski, D. J., Shepherd, G., Wingo, S., *Programming Microsoft Visual C++*, fifth Edition, Microsoft Press, 1998.

[HORT97] Horton, I., *Beginning MFC Programming*, Wrox Press, 1997.

[SCHI98] Schildt H., *MFC Programming from the Ground Up*, Osborne/McGraw-Hill, 1998.

[KAIN98] Kain, E., The MFC Answer Book, Addison-Wesley, 1998.

Le livre de Bates ainsi que le livre de Kruglinski sont des références qui traitent l'environnement de développement Visual C++. Ceux de Horton, Schildt et Kain sont dédiés au cadre de travail MFC. Celui de Shcildt offre un traitement beaucoup plus simple et facile que celui de Horton. Le livre de Kain est très utile pour ceux qui ont déjà une bonne connaissance de MFC.

Une mise en garde : Ces livres ne sont pas des références du langage C++. On ne peut pas apprendre la programmation orientée objet en lisant ces bouquins. Pire encore, les techniques de programmation présentées n'ont qu'un seul objectif : utiliser convenablement l'environnement MFC de Microsoft. Donc, il ne faut surtout pas confondre la programmation MFC avec la programmation orientée objet car ce dernier est beaucoup plus général.

PROBLÈMES

3.1 Générer, à l'aide de AppWizard, une application MDI. Compiler l'application générée et exécuter l'application.

** 3.2 Énumérer les fonctionnalités par défaut de l'application MDI générée en 3.1.

*** 3.3 Expliquer en termes pratiques la liaison d'un document à ses vues dans l'application MDI générée en 3.1. Autrement dit, comment la gestion des document templates est-elle réalisée ?

3.4 Comment peut-on créer la liste des fichiers récemment utilisés (*MRU : Most Recently Used*) dans l'application MDI créée en 3.1 ?

3.5 Comment enregistrer les fichiers de données de l'application MDI générée en 3.1 auprès de Windows Explorer ?

** 3.5 Expliquer la gestion d'une barre d'outils dans l'application MDI générée en 3.1.

3.6 Comment créer/modifier une barre de statut à une vue de l'application MDI générée en 3.1 ?

3.7 Réaliser par programmation le masquage et l'affichage de la barre d'outils de l'application MDI générée en 3.1.

3.8 Expliquer l'architecture d'impression de l'application MDI générée en 3.1.

3.9 Créer une application basée sur un panneau de dialogue (*Dialog-based*) contenant un bouton. Un clic sur ce bouton le rendera invisible. La touche ESC rendra le bouton à nouveau visible.

3.10 Créer application SDI réalisant le comportement élémentaire d'un éditeur de texte. Doter ce programme la capacité d'impression.

**** 3.11 Tenter de trouver la technique nécessaire pour changer l'icône d'une application MDI.

**** 3.12 Expliquer en détail les différents mode de traçage dans MFC (ex : `MM_TEXT`, `MM_LOENGLISH`, etc.).

CHAPITRE 4

Approche orientée objet

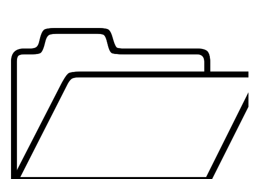
Il y a assez de clarté pour éclairer les élus et assez d'obscurité pour les humilier. Il y a assez d'obscurité pour aveugler les réprouvés et assez de clarté pour les rendre inexcusables.

— *Blaise Pascal.*

Une méthode définie une série d'étapes reproductibles pour obtenir des résultats fiables. Une méthode sert aussi à la construction des modèles à partir des éléments associés afin de représenter un système ou phénomène [MULL97]. Tout comme la plupart des méthodes scientifiques, il existe des représentations graphiques qui sont utilisées pour faciliter la manipulation des modèles. Cette représentation simplifie la communication et les échanges d'information. Le UML (*Unified Modeling Langugage*) est un outil de communication et d'échange d'information utilisé dans l'approche orientée objet. Il est indépendant du langage de programmation utilisé et encapsule les notions générales de l'analyse et conception orientées objet. La symbologie de UML accompagnera la présentation des éléments de l'approche orientée objet.

4. ÉLÉMENTS DE L'APPROCHE

Le développement logiciel est un travail intellectuel qui exige une décomposition minutieuse du travail à accomplir tout en conservant la vue d'ensemble de l'objectif final. Ce travail d'ingénierie est donc composé d'un ensemble d'étapes de « division — réunion ». La **division**, pour bien comprendre le travail à accomplir. La **réunion**, pour réaliser le travail à accomplir.



L'approche orientée objet considère un logiciel (ou système[1]) comme une entité organisée en composants qui peuvent être définis les uns par rapport aux autres. **Cette approche permet la décomposition des composants selon la nature du logiciel (ou système) et de sa fonction**. Le but de cette approche est de modéliser les propriétés statiques et dynamiques de l'environnement dans lequel les besoins ou spécifications ont été définis. Cet environnement constitue le **domaine** du problème. Ainsi, les fonctions sont représentées comme différents types de collaboration entre les objets que compose le logiciel. Lorsque le problème est bien analysé, il est alors possible de réaliser l'implantation logicielle en créant les objets qui

[1] L'apparoche orientée objet n'est pas simplement réservée qu'aux logiciels.

ont été identifiés. Le développement logiciel est donc considéré comme une intégration harmonieuse des composants de base décomposés afin de bien gérer leur complexité.

4.1 OBJETS

Un objet est défini comme une entité formée d'état et de comportement. Cette définition de l'objet assure une consistance interne et un faible couplage vers le monde extérieur. Un objet représente donc une entité complète possédant sa propre logique et règle de fonctionnement. Par contre, le rôle et la responsabilité d'un objet se dévoilent lorsqu'il est mis en communication les uns avec les autres.

Un objet est donc composé de : *i*) comportement visible; *ii*) état interne caché; *iii)* identité. On conçoit un objet comme une représentation abstraite d'un composant du système **vu par l'utilisateur**. Cette vision centrée sur l'utilisateur (*user-centric*) engendre des objets qui modélisent non pas les lois exactes régissant les composants du système mais bien une portion des connaissances d'un monde dans lequel ils évoluent. Ce monde peut obéir à des lois physiques, chimiques, biologiques, statistiques, etc. Ce monde, où évoluent les objets, peut également obéir à des règles heuristiques.

Enfin, un objet possède également une durée de vie : la durée de temps entre sa création et sa destruction. Cette considération anthropomorphique facilite la modélisation de certains phénomènes dans la vie de tous les jours.

4.1.1 NOTATION « UML »

En UML, un objet est représenté par une boîte rectangulaire. Le nom de l'objet est écrit en souligné. Prenons l'exemple du programme d'extraction des commentaires du laboratoire #1.

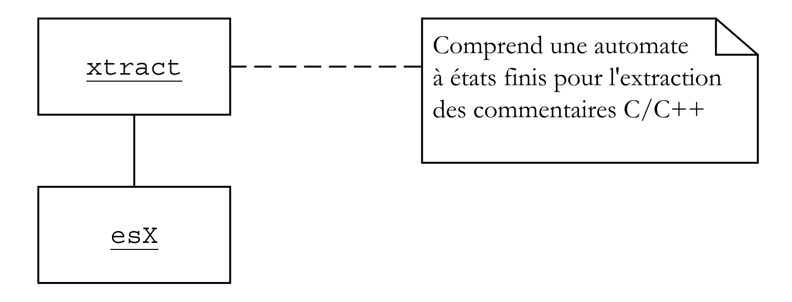


Figure 25 Notation UML pour représenter les objets.

Dans le laboratoire #1, nous avons un objet nommé `XtractComm` qui réalise l'extraction des commentaires C/C++ à l'aide d'un AEF. Un autre objet `esX` sert comme objet d'exception pour indiquer les situations anormales. La ligne pleine joignant `xtract` et `esX` signifie qu'il existe un lien entre ces deux objets. Afin de clarifier le rôle des objets, un rectangle avec le coin supérieur droit plié sert de

commentaire. La ligne pointillée joignant le rectangle de commentaire indique son appartenance. Il est possible de placer plus d'un commentaire dans un schéma UML.

Lorsque le nom des objets n'est pas encore déterminé, nous pouvons utiliser le nom des types à leur place. La notation UML utilisée pour désigner les noms génériques est montrée dans la Figure 26.

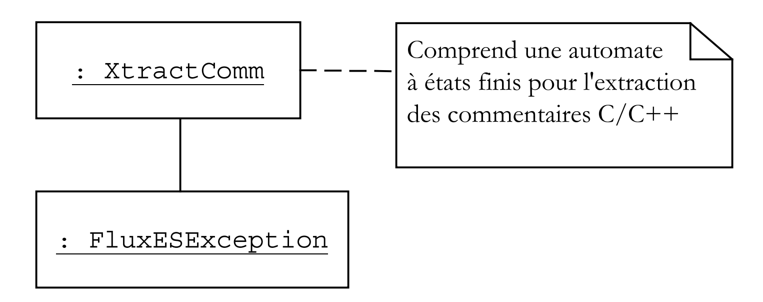


Figure 26 Notation UML utilisant les types.

Ainsi, on constate qu'il y a un objet de type `XtractComm` et un autre de type `FluxESException` impliqués dans ce schéma. Les types sont désignés par la présence du symbole : placé devant le nom.

4.2 Caractéristiques d'un objet

Pour se rappeler les caractéristiques fondamentales d'un objet il suffit de considérer la formule suivante. Ainsi,

objet = état + comportement + identité

À noter qu'un objet sans état ni comportement est possible mais cet objet n'aura aucune utilité. Par contre, tout objet doit posséder une identité (implicite ou explicite).

4.2.1 État

L'état d'un objet regroupe les valeurs de tous les **attributs** d'un objet. Un attribut est une information qui décrit l'objet. L'état d'un objet est l'ensemble des valeurs des attributs à un moment donné durant la vie de l'objet.

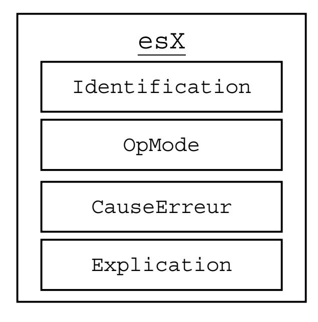


Figure 27 Un objet et ses attributs.

L'objet d'exception (de type `FluxESException`) du laboratoire #1 possède quatre attributs. La valeur de ces quatre attributs définit l'état de l'objet `esX`. Il est évident que les objets peuvent avoir différents états. Les objets passent d'un état à l'autre en fonction des stimulus internes et externes de son environnement d'exécution.

4.2.2 COMPORTEMENT

Le comportement d'un objet regroupe les capacités d'un objet et décrit les actions de l'objet. Chaque composant du comportement d'un objet est appelé une **opération**. Les opérations sont enclenchées par des stimulus externes. Voici un exemple montrant le déclenchement des opérations de l'objet `esX`.

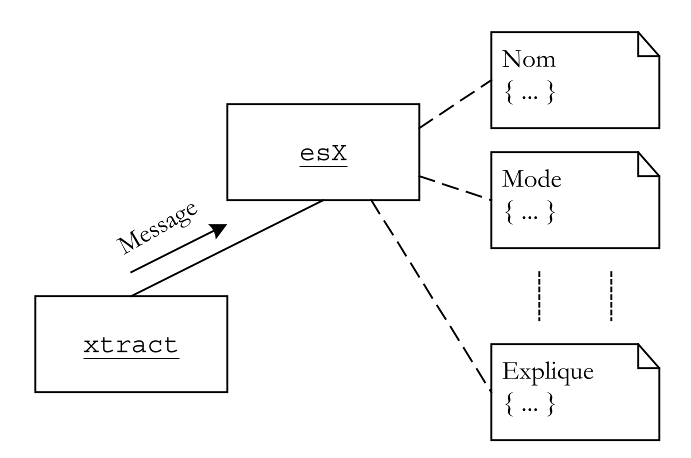


Figure 28 Déclenchement des opérations.

L'objet `xtract` est responsable du déclenchement des opérations de l'objet `esX`. Les opérations déclenchées dépendent de la nature du message envoyé par `xtract`.

L'état et le comportement sont intimement liés. En effet, le comportement d'un objet dépend de son état. L'état peut donc modifier le comportement d'un objet. Par exemple,

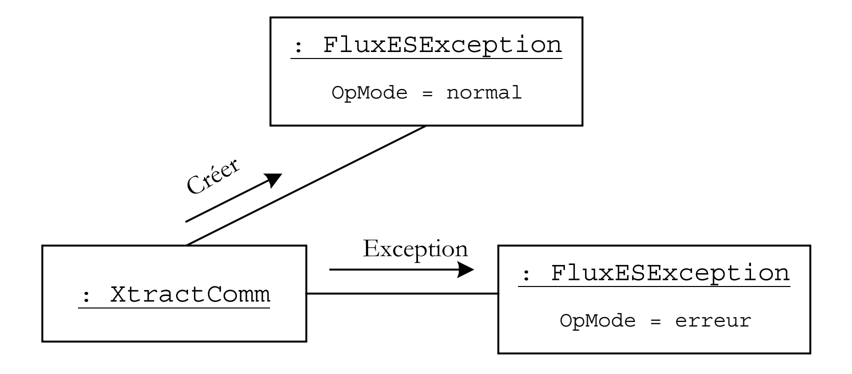


Figure 29 Modification de l'état entraîne un changement de comportement.

Après la création d'un objet de type `FluxESException` son état et son comportement sont différents par rapport à ceux qui correspondent à un événement d'exception.

4.2.3 IDENTITÉ

L'identité d'un objet permet de faire la distinction entre les objets de même type. Cette distinction peut être implicite ou explicite. Le nom de l'objet est une identité implicite. Ainsi, `xtract` et `esX` constituent l'identité implicite des objets de type `XtractComm` et `FluxESException` respectivement. Pour réaliser l'identité explicite des objets, nous devons obtenir une information particulière dans le domaine du problème. Cette information doit pouvoir identifier d'une manière non équivoque les objets. Par exemple, les objets représentant les étudiants de l'École peuvent être identifiés par leur code permanent, les véhicules d'automobile par leur numéro de série, etc.

Dans le UML, il n'y a pas de symbole réservé pour désigner l'identité des objets. Cette information est souvent traitée comme un attribut.

4.3 CONSIDÉRATIONS D'IMPLANTATION

Dans la présentation des sous-sections précédentes nous avons traité les objets d'une manière générale sans prendre en considération leurs implantations pratiques. Nous allons effectuer, dans cette section, un survol de quelques considérations pratiques reliées à l'implantation des objets.

4.3.1 PERSISTANCE DES OBJETS

Le terme **persistance** signifie la permanence des objets dans le temps. Pour réaliser la persistance des objets, nous devons disposer d'un mécanisme d'entreposage capable de maintenir en mémoire l'état et le comportement des objets. Autrement dit, il est nécessaire de conserver, dans une mémoire non volatile, les objets entiers. Le mécanisme de la persistance des objets implique donc l'enregistrement et le chargement des objets à partir de la mémoire non volatile.

Curieusement, les objets ne sont pas, par définition, persistants et les langages de programmation orientée objet n'offrent pas de capacité intrinsèque concernant la persistance des objets. La sérialisation[2] est l'une des solutions à ce problème. Les outils de développement tels le Visual C++ de Microsoft et le C++ Builder de Borland offrent des solutions ajoutées pour palier à ce problème.

[2] Voir l'aide en-ligne de Visual C++ pour comprendre ce qu'est la sérialisation.

4.3.2 DIFFUSION DES OBJETS

La diffusion des données (*Data broadcasting*) est un concept bien étudié. De nos jours, on peut facilement obtenir une information à distance via l'Internet. Or, il est encore difficile de parler d'une diffusion des objets. L'une des méthodes envisagées consiste à analyser le contenu d'un objet puis transférer la description de l'objet sous forme de données. À la réception, la description reçue de l'objet est ensuite utilisée pour recréer l'objet lui-même. Il y donc une opération de « clônage » impliquée dans la diffusion des objets. Le problème majeur de cette technique réside dans le fait que le récepteur doit pouvoir reconnaître l'ensemble des objets à transmettre. Sans cette connaissance a priori des objets, il est très difficile de les identifier et de les recréer.

4.3.3 OBJETS PROXY

Un objet « proxy » est un objet qui se comporte exactement comme un autre objet avec lequel il est synchronisé. Les objets synchronisés peuvent être liés par un lien de communication. Ainsi, un objet proxy permet la manipulation d'un objet à distance. Il s'agit donc une alternative de la diffusion des objets. D'un point de vue utilisateur, toute la complexité de communication est cachée par les objets proxy.

4.4 INTERACTIONS DES OBJETS

Dans l'approche orientée objet, la fonctionnalité d'un logiciel est réalisée par les interactions entre les objets. On peut identifier trois catégorie de comportement général dans ces interactions : *i*) serveur; *ii*) client; *iii*) agent.

Un client est un objet qui initie une interaction. Dans la Figure 30, l'objet `xtract` est un client et on l'identifie comme étant un objet actif et passe le contrôle à `esX` via un message. Dans la même logique, un serveur est la cible des messages. Un objet serveur est un objet passif puisqu'il attend l'arrivée des messages en provenance des autres objets. Enfin, un agent combine les caractéristiques d'un client et d'un serveur. Les objets agents interagissent en tout moment avec les autres objets soit de leur propre volonté soit d'une réaction en fonction des stimulus externes. Souvent, un agent est interposé entre un client et un serveur afin de rediriger les requêtes du client. Cet arrangement est motivé par le fait qu'un client peut ne pas connaître le serveur qui traitera ses requêtes. Le rôle de l'agent, dans ce cas, sert à aiguiller les requêtes du client vers le bon serveur. La figure montre un arrangement possible de ces comportements généraux des objets.

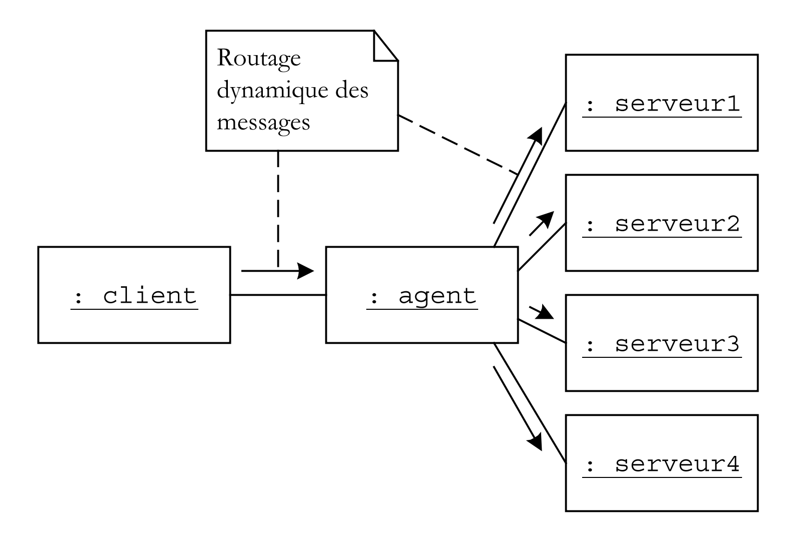


Figure 30 Liens entre un client, un agent et des serveurs.

4.5 MESSAGES ET LES OBJETS

Un message est l'unité de base des interactions entre objets. Les messages sont des ingrédients essentiels pour l'établissement d'une communication entre deux objets. Puisque la fonctionnalité d'une application dépend des échanges entre les objets, on voit bien que les messages jouent un rôle très important dans l'approche orientée objet. La réalisation concrète des messages peut prendre différentes formes. Ainsi, un appel de fonction, une interruption, un événement, un datagramme peuvent être considérés comme des formes de messages. La figure Figure 31 donne la description complète d'un message selon la notation UML.

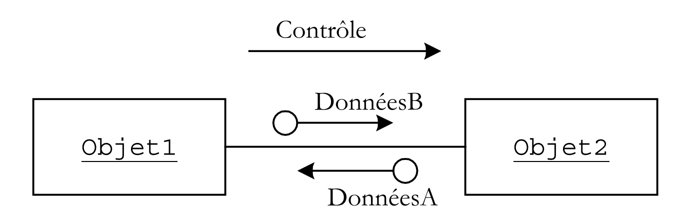


Figure 31 Description complète d'un message envoyé de `Objet1` à `Objet2`.

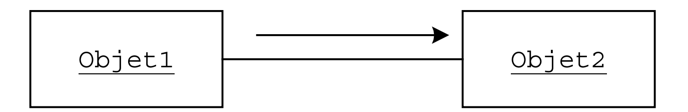
La direction du message est indiquée par la ligne fléchée. Le transfert des données est indiqué par une ligne fléchée débutant par un cercle.

Nous pouvons également distinguer plusieurs types de message : *i*) constructeur; *ii*) destructeur; *iii*) sélecteur; *iv*) modificateur; *v*) itérateur. Les messages de type constructeur servent à créer les objets. Comme son nom l'indique, un message destructeur déclenche la destruction des objets. Les messages sélecteurs servent à obtenir des informations des objets. Les messages modificateurs sont utilisés pour modifier les attributs des objets. Enfin, les messages itérateurs servent à obtenir un objet parmi un ensemble d'objets.

4.5.1 MESSAGES DE SYNCHRONISATION

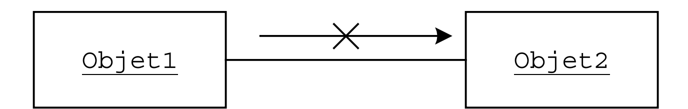
Les messages de synchronisation permettent, la réalisation des accès ordonnés d'un objet partagé ou d'un objet encapsulant une ressource critique. Il existe également plusieurs types de messages de synchronisation.

- **Diffusion simple** : Ce type de message de synchronisation est utilisé dans le cas où il n'y a qu'un seul objet actif à la fois. Le contrôle est passé d'un objet actif à un objet passif.



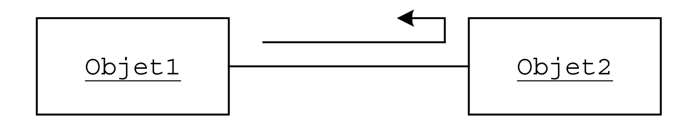
Diffusion simple

- **Diffusion synchrone** : Un message synchrone déclenche une opération de l'objet récepteur seulement si ce dernier est prêt à recevoir le message. L'objet expéditeur de message est bloqué jusqu'à ce que le récepteur accepte le message.



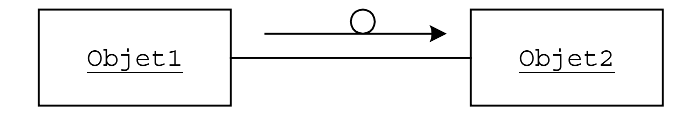
Diffusion synchrone

- **Rendez-vous** : L'objet récepteur du message doit d'abord se placer en mode bloqué en attente d'un message. C'est l'inverse de la diffusion synchrone où c'est l'objet expéditeur qui est bloqué.



Diffusion rendez-vous

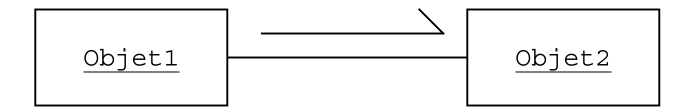
- **Message temporisé** : Un message temporisé met l'objet expéditeur en état bloqué pendant une période de temps déterminée afin de permettre à l'objet récepteur de lui signifier la réception du message. Après la période d'attente, l'objet expéditeur est libéré d'une manière inconditionnelle.



Diffusion temporisée

- **Message asynchrone** : La transmission d'un message asynchrone ne bloque pas l'objet expéditeur ou l'objet récepteur. L'accusé de réception d'un message

asynchrone peut être reçu par l'expéditeur en un moment quelconque après la transmission du message.



Diffusion asynchrone

4.6 REPRÉSENTATION DES INTERACTIONS

Nous avons vu que l'interaction entre les objets s'effectue par le biais des messages. Deux diagrammes UML sont disponibles pour représenter, d'une manière concise, l'envoie et la réception des messages.

Le diagramme de collaboration indique les interactions entre objets. **L'emphase de ce diagramme est mise sur les relations entre les objets**. La figure suivante montre un diagramme de collaboration comprenant trois types d'objet.

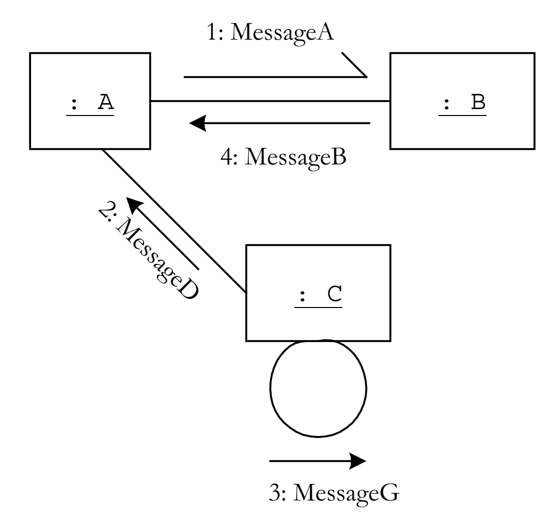


Figure 32 Diagramme de collaboration.

Dans un diagramme de collaboration, les messages sont identifiés par un numéro afin d'y donner un ordre chronologique. Noter que le message 3 (l'objet de type C) est un message envoyé à soi-même. Il s'agit là d'une notation UML représentant les activités internes d'un objet.

Le problème avec ce genre diagramme est que le nombre de messages peut rendre sa lecture très difficile. En effet, l'encombrement des messages peut masquer facilement le sens ou le but des interactions décrites par le diagramme.

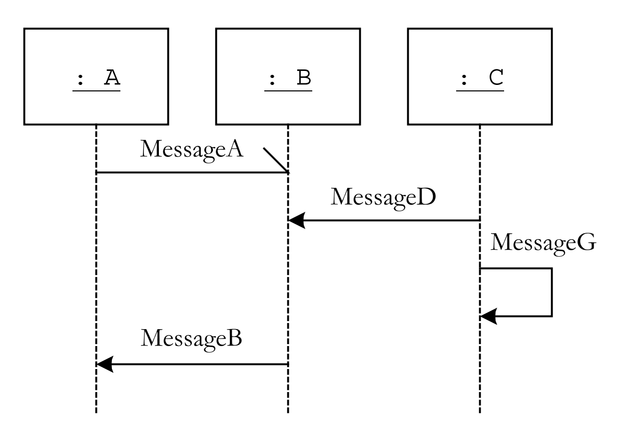
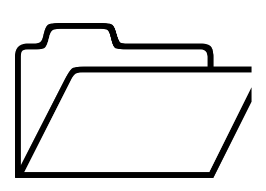


Figure 33 Diagramme de séquence.



C'est pour cette raison que nous utilisons églament un autre diagramme appelé diagrame de séquence. **Un diagramme de séquence contient les mêmes informations qu'un diagramme de collaboration**. Un diagramme de séquence met en valeur les messages transmis entre les objets. Par contre, les liens qui existent entre les objets ne sont pas présentés aussi explicitement que le diagramme de collaboration. Le diagramme de séquence de la Figure 33 présente les mêmes informations que le diagramme de collaboration de la Figure 32.

4.7 CLASSES

Les humains ont érigé des systèmes de pensée afin de saisir la complexité du monde sensible qui les entoure. La décomposition et la classification sont des techniques qui aident à la compréhension des phénomènes complexes. La décomposition permet la réduction de l'ensemble en éléments facilement assimilables. La classification permet le regroupement et la catégorisation des phénomènes semblables. Ces deux techniques et bien d'autres s'opèrent à un niveau élevé d'idéalisation : le niveau d'abstraction des idées.

Le processus d'abstraction est une opération consciente qui consiste à identifier les caractéristiques communes d'un ensemble d'éléments et qui aboutit à une description concise de ces caractéristiques. De plus, le processus d'abstraction est une opération équivoque : son point de départ dépend du point de vue adopté. Ainsi, une entité sensible peut être vue à travers différentes abstractions. À cause de cette amphibologie du processus d'abstraction, il est très important de définir les critères d'abstraction qui représentent bien les objectifs visés.

Dans l'approche orientée objet, la description concise des caractéristiques résultant de l'abstraction est appelée une **classe**. Une classe décrit donc les caractéristiques générales d'un ensemble d'objets. Ainsi, chaque objet appartient à une classe. La création d'un objet à partir de sa classe est appelée une **instanciation**. C'est pour cette raison que l'on dit un objet est une **instance** d'une classe.

4.7.1 NOTATION UML

Une classe, tout comme un objet, est représentée par un rectangle. Cependant, ce rectangle est subdivisé en trois parties. La Figure 34 présente une classe selon la notation UML.

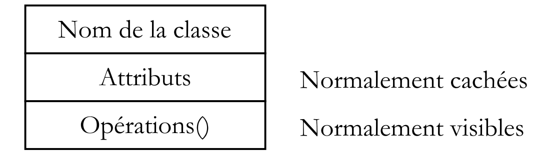


Figure 34 Notation UML d'une classe.

Les attributs d'une classe sont normalement invisibles à l'extérieur de la classe. Ainsi, pour obtenir ou modifier la valeur des attributs, il est nécessaire de recourir aux opérations disponibles. Donc, les opérations d'une classe sont normalement visibles dans l'environnement externe.

Il arrive parfois que les classes ne disposent pas d'attributs ou d'opérations. Dans ce cas, la notation UML ressemblera à celle de la Figure 35.

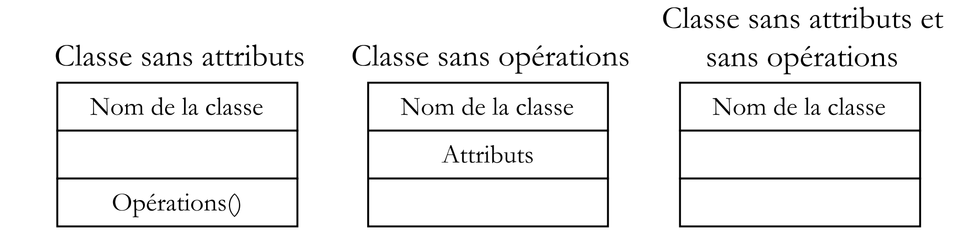


Figure 35 Notation UML pour les classes sans attributs et/ou sans opérations.

Les classes sans attributs et/ou sans opérations sont plutôt rares. On les utilise surtout pour simplifier le travail. Ainsi, lorsque le contexte est évident, il est possible d'éviter la répétition en n'écrivant que le nom de la classe.

4.7.2 DESCRIPTION DES CLASSES

La description des classes comprend deux parties : *i)* interface; *ii)* implantation. L'interface d'une classe décrit le domaine de définition et les propriétés des instances de cette classe. Elle correspond exactement à la notion des types des langages de programmation orientée objet. L'implantation d'une classe décrit le processus de réalisation de la spécification. Elle contient le corps des opérations et les données nécessaires à leur fonctionnement.

D'une façon pragmatique, une classe est en relation contractuelle avec les autres classes. Une classe offre les services publiés dans son interface et les autres classes acceptent d'utiliser ces services sans outre passé la confidentialité de ceux-ci.

La séparation entre l'interface et l'implantation est un point important dans l'approche orientée objet. En effet, les caractéristiques intéressantes d'une classe sont décrites dans son interface. Les détails et considérations techniques sont confinés dans son implantation. Il s'agit là de la notion d'**encapsulation**. L'encapsulation protège les données d'une classe contre les accès inopportuns. Elle diminue également la force du couplage entre la classe et son modèle. Un utilisateur n'a pas à connaître les détails d'implantation d'une classe mais seulement les spécifications de son interface.

Normalement, la valeur des attributs d'un objet n'est pas accessible directement par un autre objet. On dit que les attributs sont masqués à l'intérieur d'un objet. L'interaction entre les objets s'opère en activant différentes opérations déclarées dans l'interface. Ainsi, les opérations de l'interface d'une classe sont accessibles à d'autres classes. Pour donner un plus grand contrôle (et de sécurité) sur l'accessibilité des opérations, plusieurs niveaux de protection sont généralement disponibles. Par exemple, le C++ offre trois niveaux d'accessibilité :

- **Niveau privé** : Il s'agit du niveau de protection le plus élevé. Seuls les objets de même classe et les fonctions et les objets déclarés amis peuvent y accéder.

- **Niveau protégé** : Il s'agit du niveau de protection intermédiaire. Seuls les objets de même classe et les objets de classes dérivées peuvent y accéder.

- **Niveau public** : Il s'agit du niveau de protection le plus faible. L'effet de l'encapsulation est éliminé. Les opérations et les attributs placés dans ce niveau sont accessibles par tous.

Les symboles UML utilisés pour représenter ces trois niveaux de protection sont : *i)* `+` (niveau public), *ii)* `#` (niveau protégé); *iii)* `–` (niveau privé). La Figure 36 donne un exemple utilisant cette symbologie de UML.

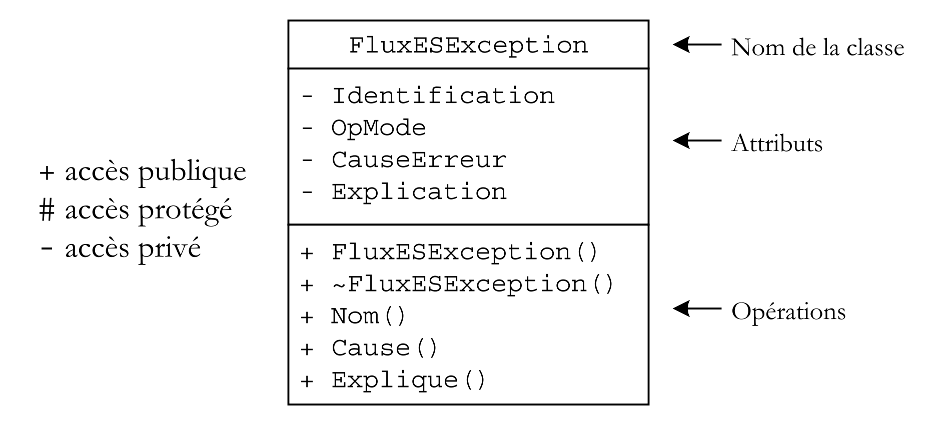
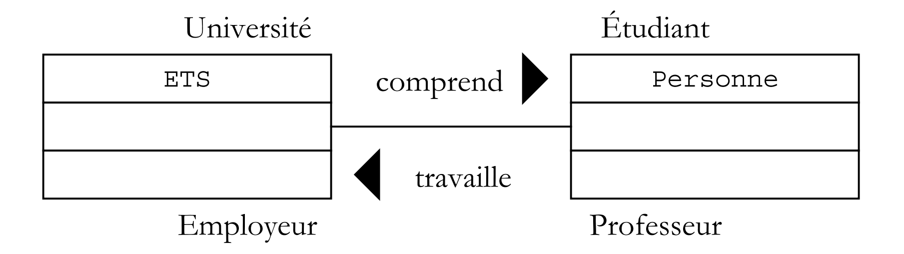


Figure 36 Description d'une classe avec identification des niveaux d'accès.

4.7.3 RELATIONS ENTRE LES CLASSES

Nous pouvons établir des relations qui lient les classes entre elles. Tout comme dans le monde sensible, ces relations ne sont pas identiques. Plutôt, elles peuvent prendre différentes formes et revêtent différentes significations. Voici les différentes relations généralement admises dans l'approche orientée objet :

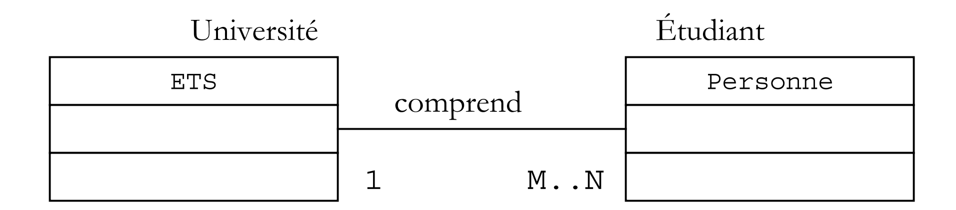
Association : Une association des classes est équivalente à un lien entre les objets. La relation d'association exprime une abstraction du lien qui existe entre les objets. Normalement, nous devons spécifier le rôle des classes et la nature de cette association.



Pour indiquer le nombre d'instances (objets) impliqué dans une relation, utilisons la notation suivante. Le Tableau 14 résume cette information appelée la **multiplicité**.

Symbole	Signification
1	Une seule instance.
0..1	De zéro à 1 instance.
M..N	De M à N instance. M > N, M et N sont des entiers non négatifs.
*	De 0 à un nombre quelconque d'instance.
0..*	Identique à *.
1..*	De 1 à un nombre quelconque d'instance.
M..*	De M à un nombre quelconque d'instance. M est un entier non négatif.

Tableau 14 Symboles utilisés pour indiquer la multiplicité des instances.



N > M, N est la capacité maximale d'accueil de l'École.

Figure 37 Utilisation de la multiplicité dans un diagramme de classes.

Reprenons l'exemple de la Figure 37 où la relation entre la classe `ETS` et la classe `Personne` a été exprimée. L'information de la multiplicité est indiquée et on peut

comprendre qu'il y aura une instance de la classe `ETS` et M à N instances de la classe `Personne`.

À remarquer que, si le contexte est clair, il n'est pas nécessaire d'indiquer la direction de la relation ni même le rôle des classes.

Agrégation : Une relation d'agrégation est une relation plus forte que celle de l'association. L'agrégation permet l'expression des relations de type : « maître et esclave », « une partie de », « composé de », etc. Ainsi, dans une relation d'agrégation, l'une des classes est plus importante que l'autre. En termes mathématiques, une relation d'agrégation est une relation transitive, asymétrique et peut être réflexive.

Soit R une relation binaire définie sur l'ensemble A. La relation R est **transitive** si et seulement si $\forall\ x, y, z \in A$, si $(x, y) \in R$ et $(y, z) \in R$ alors $(x, z) \in R$. La relation R est **asymétrique** si $\forall\ x, y \in A$, $(x, y) \in R$ alors $(y, x) \notin R$. Enfin la relation R est **réflexive** ssi $\forall\ x, y \in A$, $(x, x) \in R$

L'exemple de la figure montre un diagramme de classes exprimant la relation qui existe entre un parent et ses enfants. À noter que les enfants et leur parent sont issus de la classe `Personne`. L'information de multiplicité est appliquée en fonction du domaine d'intérêt. Ainsi, ce diagramme exprime l'ensemble des enfants orphelins (0 parent) ou des enfants vivant dans une famille ayant 1 ou 2 parents.

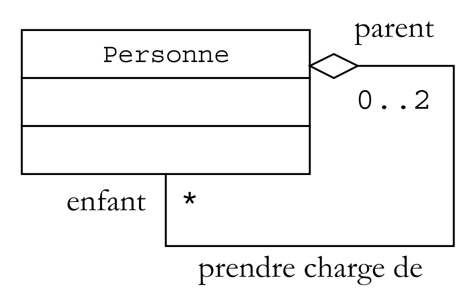


Figure 38 Relation d'agrégation exprimant une certaine relation entre les enfants et leurs parents.

La relation d'agrégation de la Figure 38 est réflexive puisque la classe Personne peut jouer le rôle de parent et celui de l'enfant. Elle est également asymétrique puisque les enfants ne peuvent (en théorie) prendre charge de leurs parents. Le symbole ◇ est utilisé pour désigner une relation d'agrégation.

Pour exprimer une relation d'agrégation plus forte où l'élimination d'un composant entraîne l'élimination de l'ensemble, nous devons utiliser une forme d'agrégation appelée **composition**. Cette relation est plus forte que celle de l'agrégation standard puisqu'il existe une dépendance directe entre les classes. Dans l'exemple de la Figure 39, un enfant peut ne pas avoir de parent. De même, les parents peuvent ne pas avoir d'enfants à leur charge. Dans une relation de composition, les classes existent

et supportent mutuellement dans leurs fonctions. Le symbole ◆ est utilisé pour exprimer une relation de composition.

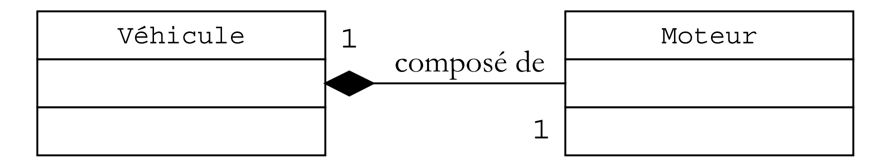


Figure 39 Relation de composition.

4.7.4 HIÉRARCHIE DES CLASSES

L'établissement d'une hiérarchie des classes a pour objectif la classification des objets en niveaux d'abstraction plus ou moins élevés. Une technique généralement utilisée dans l'approche orientée objet consiste à adopter deux points de vue afin de réaliser cette classification. La **généralisation** et la **spécialisation** sont ces points de vue.

La généralisation consiste à regrouper les éléments communs d'un ensemble de classes en une seule classe générale appelée surclasse (classe de base en C++). Les éléments communs sont généralement les attributs, les opérations des classes intéressées. Ainsi, une surclasse est une abstraction de ses sous-classes.

L'exemple ci-dessous montre une généralisation appliquée à un certain type de véhicule. Les lignes fléchées pointent vers des classes de plus en plus générales.

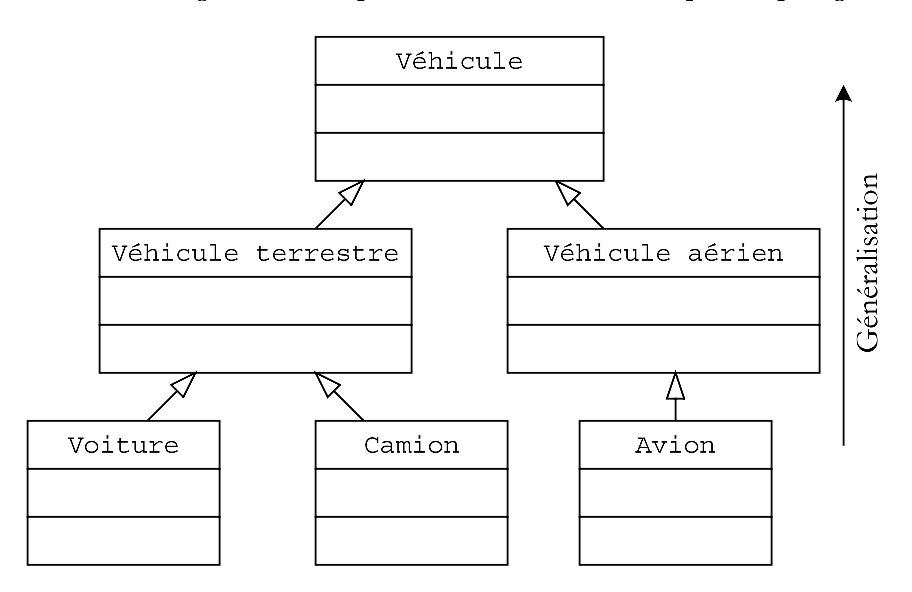
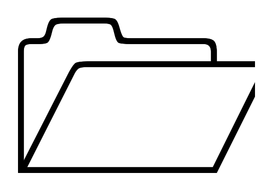


Figure 40 Généralisation des classes.



Ainsi, la classe `Véhicule` est une surclasse des classes `Véhicule terrestre` et `Véhicule aérien`. Ces sous-classes de `Véhicule` sont, à leur tour, des surclasses de `Voiture`, `Camion` et `Avion` respectivement. **La relation qui existe entre les sous-classes et leur surclasse est de type « est-un » ou « est-une-sorte-de »**. Par conséquent, la classe `voiture` de la Figure 40 est un véhicule terrestre et ce dernier

est une sorte de véhicule. La spécialisation permet l'ajout de nouvelles caractéristiques à un ensemble d'objets qui n'ont pas été identifiées auparavant dans la hiérarchie des classes. La spécialisation permet donc l'extension des capacités d'une manière cohérente à un ensemble d'objets.

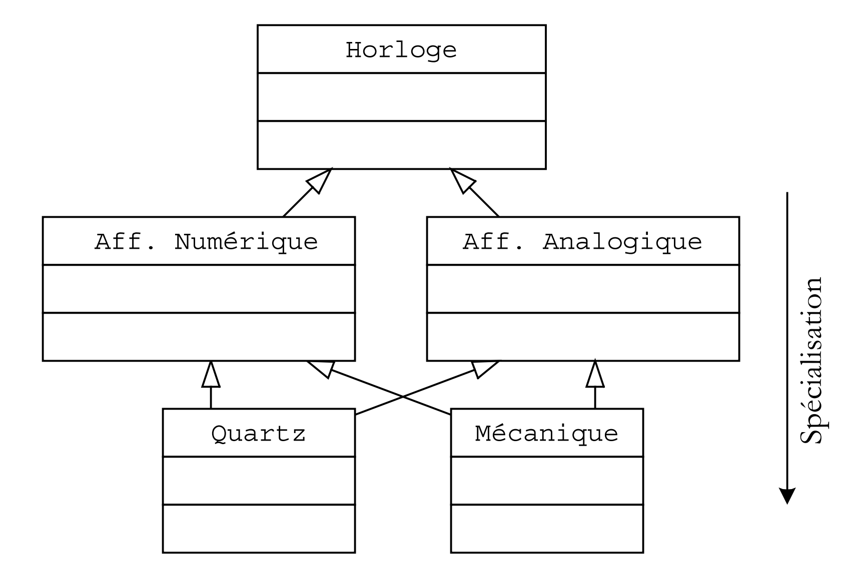


Figure 41 Spécialisation des classes.

La généralisation et la spécialisation sont deux points de vue opposés. Dans la pratique, ces deux points de vue sont utilisés en même temps pour étendre la hiérarchie des classes. Normalement, la généralisation est appliquée au tout début de la définition du domaine de l'application. La spécialisation est exercée lors de la phase de conception de l'application afin d'étendre les capacités des classes.

Il est très difficile d'effectuer la classification par généralisation et spécialisation. Il n'existe pas de règles infaillibles qui puissent mener vers une classification parfaite. Néanmoins, il est possible de dégager quelques points importants que l'on doive surveiller lors de l'application du processus de généralisation et de spécialisation.

- Une surclasse renferme les caractéristiques communes de ses sous-classes. Ainsi, il est important de bien recenser les caractéristiques souhaitables des sous-classes avant de créer la surclasse.

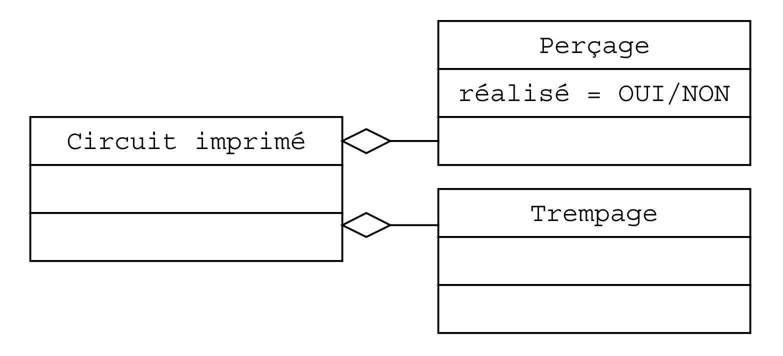
- Il ne faut pas surcharger la surclasse avec des détails qui ne sont profitables qu'à quelques sous-classes seulement.

- Une surclasse ne doit pas différer de ses sous-classes que par des caractéristiques mineures. Dans ce cas, unir les classes et différencier les objets par la valeur de ces caractéristiques. Par exemple, il est un gaspillage de créer une surclasse et des sous-classes afin de distinguer la couleur des voitures.

- Une surclasse doit pouvoir prévoir l'évolution possible de ses sous-classes. Ainsi, une surclasse peut se doter d'opérations qui anticipent leur utilisation éventuelle.

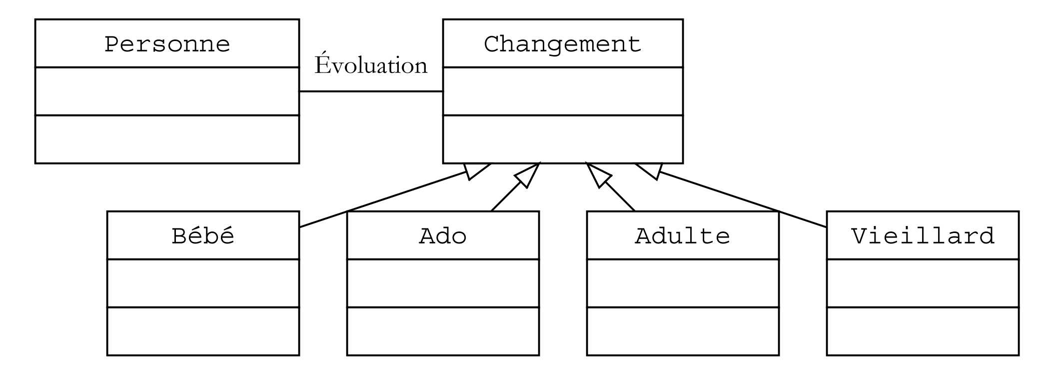
Par exemple, une surclasse peut posséder les opérations de sauvegarde même si elles ne sont pas exigées explicitement.

- La généralisation n'est pas une technique convenable pour modéliser le changement dynamique. Par exemple, un produit est réalisé en appliquant des transformations sur le matériel brut. La création d'une surclasse et de ses sous-classes pour représenter le matériel brut et les transformations appliquées n'est pas une approche conseillée. Plutôt, on peut user de la relation d'agrégation pour représenter ce type de changement. Ainsi,



On aurait pu représenter les transformations par des attributs de la classe `Circuit imprimé`. Par contre, s'il existe un nombre de procédés différents pour réaliser ces transformations alors il serait plus convenable de créer des relations d'agrégation.

- Une autre technique pour représenter le changement dynamique consiste à utiliser une relation d'association entre une classe et ses éléments mutables. Ainsi, on peut signifier les étapes de la vie d'une personne par :



4.7.5 HÉRITAGE

L'héritage est une façon pratique de réaliser la classification. Dans la programmation orientée objet, l'héritage est l'outil principal pour la propagation des caractéristiques des classes (de la surclasse vers ses sous-classes). Le concept d'héritage sert aussi de construction. Par exemple, la classe collection `vecteur` peut être construite à partir d'une liste chaînée. La classe `vecteur` est une sous-classe de la classe `ListeChainee`. La classe `vecteur` hérite les fonctionnalités de `ListeChainee` et apporte en elle des capacités nouvelles d'un vecteur linéaire. **Il est clair que le**

concept d'héritage joue le rôle de propagation des caractéristiques (classification) et de la construction des classes et ce, en fonction du point de vue que l'on a adopté.

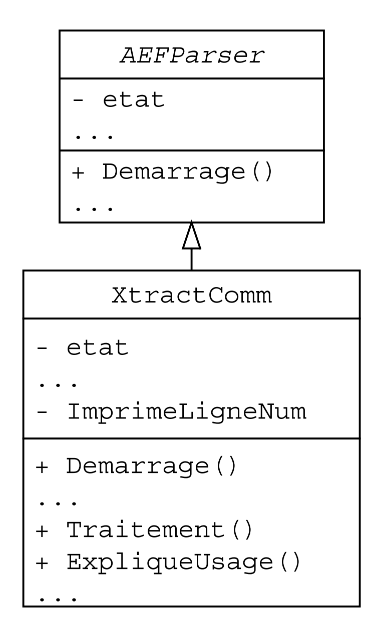


Figure 42 Effet de l'héritage.

L'exemple de la Figure 42 est tiré des classes de laboratoire #1. La classe `AEFPaser` est la surclasse (classe de base) de la classe `XtractComm`. Cette dernière hérite les attributs et les opérations de `AEFParser` tout en ajoutant ses propres attributs et opérations.

Il est à noter que l'héritage peut être multiple. C'est-à-dire, une sous-classe peut hériter les caractéristiques de plus d'une surclasse. Cependant, l'héritage multiple produit de l'ambiguïté qui est très difficile à résoudre. Pour cette raison, nous ne traiterons pas de la possibilité de l'héritage multiple dans ce cours.

Nous pouvons utiliser le **principe de substitution** pour déterminer si l'héritage envisagé est convenable pour réaliser une classification des objets. Ce principe s'énonce comme ceci :

« ***Il doit être possible de substituer n'importe quelle instance d'une surclasse par n'importe quelle instance d'une sous-classe sans changer la sémantique d'un programme écrit en termes de la surclasse***. ».

En effet, la généralisation des classes implique que les caractéristiques de la surclasse sont incluses dans les caractéristiques des sous-classes. Cependant, le langage de programmation C++ permet la redéfinition des opérations par la surcharge des

fonctions membres. Il est alors possible d'introduire de l'inconsistance entre l'interface d'une classe parente et l'interface de ses enfants. C'est la responsabilité du programmeur de vérifier la satisfaction de ce principe de substitution. Le principe de substitution doit être respecté si nous désirons réaliser le **polymorphisme**.

4.7.6 POLYMORPHISME

Le terme polymorphisme signifie un élément qui peut prendre plusieurs formes. Par exemple, l'argent peut être sous forme de papier ou de monnaie. D'un point de vue orienté objet, le polymorphisme est un concept dans lequel un nom d'objet peut désigner des instances de différentes classes dans une même hiérarchie de classes.

En termes pratiques, le polymorphisme des classes est appliqué à des opérations. Dans ce contexte, la surclasse est dotée d'opérations dont le mécanisme représente une abstraction qui sera réalisée par ses sous-classes. Les sous-classes, quant à elles, héritent les caractéristiques de sa surclasse et réalisent localement l'implantation du mécanisme à l'aide de ses opérations. Par exemple, une figure géométrique possède un attribut appelé `coordonnées`. Nous pouvons construire une hiérarchie de classes partant de ce point de départ. Les figures géométriques sont également dotées d'une opération appelée `Affichage()`. Évidemment, le mécanisme d'affichage peut différer d'une figure à l'autre. Ainsi, le polymorphisme des opérations est exprimé par un mécanisme général et abstrait instauré dans la surclasse.

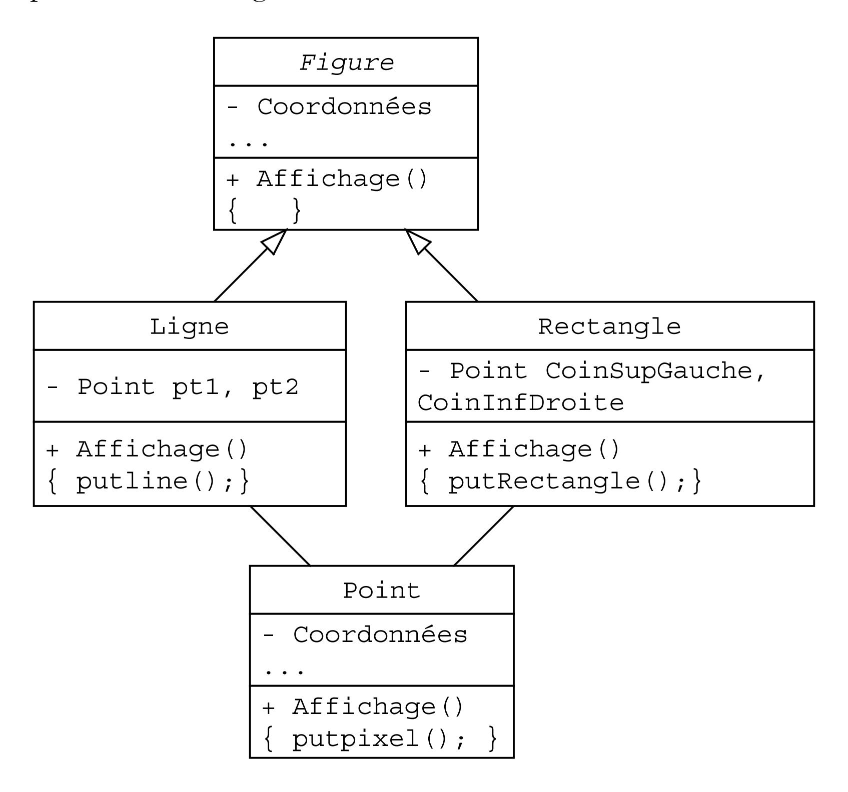


Figure 43 Hiérarchie de classes possédant le polymorphisme des opérations.

Les sous-classes `Point`, `Ligne` et `Rectangle` disposent d'une opération `Affichage()`. La capacité de l'affichage des figures géométriques est instaurée dans

la surclasse `Figure`. Chaque sous-classe réalise différemment le mécanisme de l'affichage.

À noter que le polymorphisme s'opère toujours à travers l'interface d'une surclasse. Si la classe `Figure` de la Figure 43 ne disposait pas, dans son interface, l'opération `Affichage()` alors on ne peut réaliser l'effet voulu dans les sous-classes. Heureusement, ce genre d'erreur peut être repéré par le compilateur puisque la description ou la syntaxe pour la réalisation du polymorphisme est souvent de nature statique (par l'utilisation d'un mot de clé). D'un autre côté les sous-classes peuvent, par oubli ou par exprès, ne pas réaliser l'implantation des opérations polymorphiques. Dans ce cas, c'est l'opération générique de la surclasse qui sera utilisée. Par exemple, nous pouvons ajouter une sous-classe `Ellipse` à la classe `Figure` et ne pas implanter l'opération `Affichage()` pour Ellipse. La syntaxe demeura correcte et le compilateur ne peut déceler d'erreur. Cependant, le fait d'afficher une ellipse par l'opération de la surclasse `Figure` est probablement une erreur sémantique.

Il est possible d'utiliser le concept de **classe abstraite** pour forcer la réalisation correcte du concept de polymorphisme. Une classe abstraite est la généralisation du mécanisme abstrait des opérations. Elle est le résultat d'un haut niveau d'abstraction dans lequel toute la classe (et non pas seulement ses opérations) représente une idée ou un concept en devenir. La concrétisation de cette idée ou concept est reléguée aux sous-classes. Une classe abstraite est représentée par le nom écrit en italique (voir Figure 44). De plus, **on ne peut instancier un objet à partir d'une classe abstraite**. Une classe abstraite sert uniquement de surclasse à d'autres classes. Les sous-classes issues d'une classe abstraite doivent absolument réaliser l'implantation des opérations désignées. En C++, la création d'une classe abstraite est automatique. Dès qu'une classe contient une fonction membre purement virtuelle[3], elle est automatiquement considérée comme une classe abstraite.

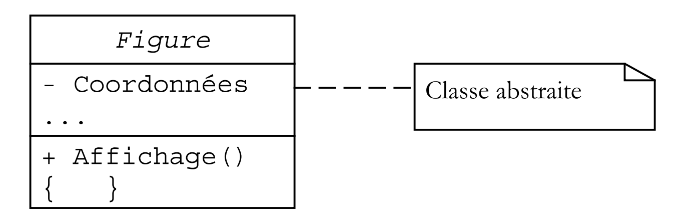


Figure 44 Notation UML pour désigner une classe abstraite.

Le mécanisme général pour le déclenchement des opérations polymorphiques est simple à comprendre. Nous utiliserons la hiérarchie de classes ci-dessous pour démontrer ce mécanisme.

[3] voir chapitre 1 pour connaître la syntaxe d'une fonction membre purement virtuelle.

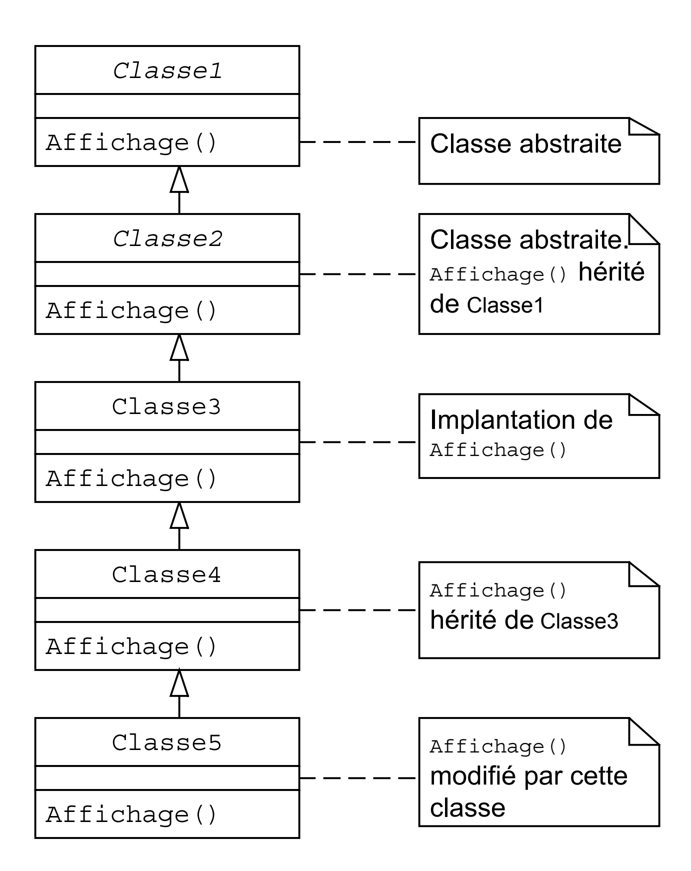


Figure 45 Hiérarchie de classes pour la discussion des opérations polymorphiques.

Définissons une classe `Client` dont les instances (objets) peuvent communiquer avec les objets des classes concrètes[4] `Classe3`, `Classe4` ou `Classe5`. Observer bien les liens du diagramme d'objets suivant.

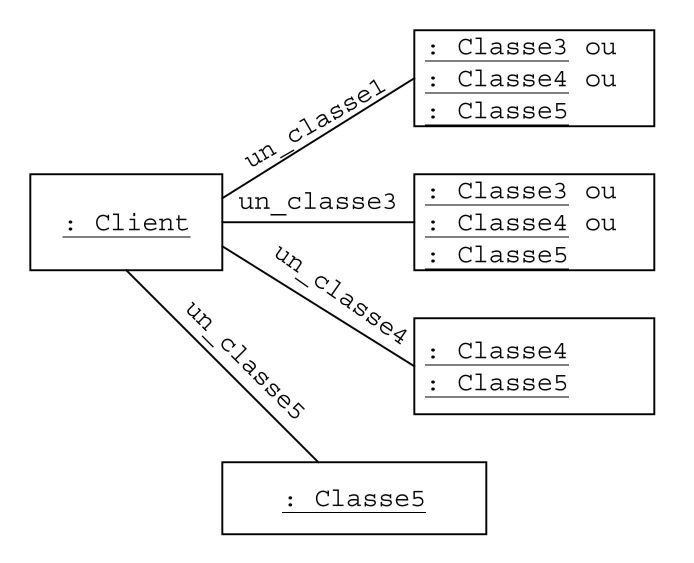


Figure 46 Relation entre les objets de la classe `Client` et les objets de `Classe3`, `Classe4` et `Classe5`.

[4] Par opposition à des classes abstraites.

Le lien un_classe1 est polymorphique et il peut désigner tout objet des classes concrètes (Classe3, Classe4 ou Classe5). De même pour le lien un_classe4. Les objets de la classe Client ayant ce lien peuvent manipuler une instance des classes Classe4 et Classe5. **Les objets de la classe Client peuvent donc manipuler une instance de ces classes à travers l'interface d'une surclasse**. Par exemple, le message Affichage() (voir la hiérarchie la Figure 45) peut être envoyé à un objet de la classe Classe5 par le lien polymorphique un_classe3. Le code exécutable de l'opération Affichage() est fouillé lors de son invocation en parcourant l'arbre d'héritage des classes. La classe Classe5 possède une implantation de l'opération Affichage() et c'est cette dernière qui sera exécutée.

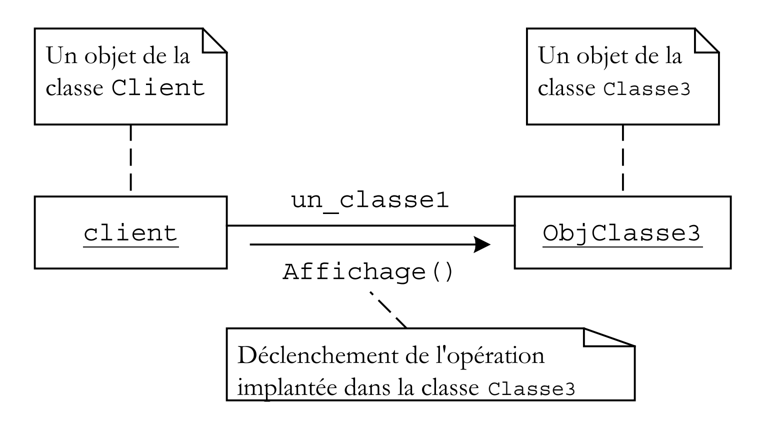


Figure 47 Invocation d'une opération de la classe Classe3.

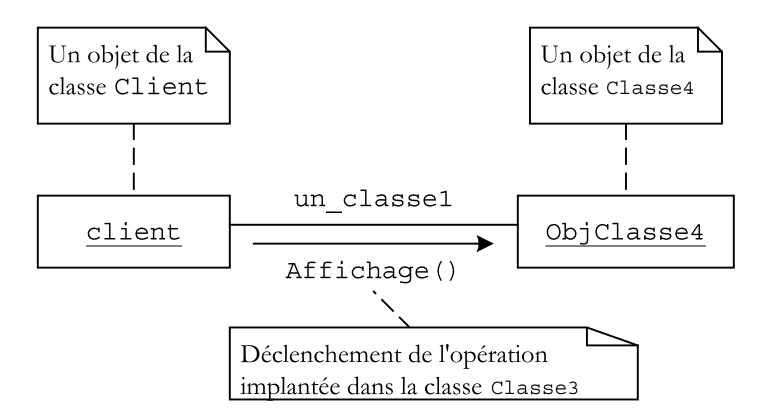


Figure 48 Invocation de la même opération même si l'objet est de la classe Classe4.

De même, il est possible d'envoyer le message Affichage() à une instance de la classe 3 via le lien un_classe1 tel qu'illustré dans la Figure 48. Cependant, le même message Affichage() envoyé à un objet de Classe4 provoque le déclenchement de l'opération implantée dans la Classe3.

4.8 EXEMPLE D'APPLICATION

Cette sous-section présente un exemple informel montrant l'application des diagrammes UML dans la modélisation. Cet exemple est un problème impliquant la modélisation et la simulation d'un système d'ascenseurs [HTTP01].

Description du problème

Le campus principal de l'École de technologie supérieure possède trois ascenseurs A_1, A_2 et A_3. L'emplacement de ces ascenseurs est montré dans la Figure 49. Les étages de l'édifice abritant le campus principal sont également identifiés dans cette figure.

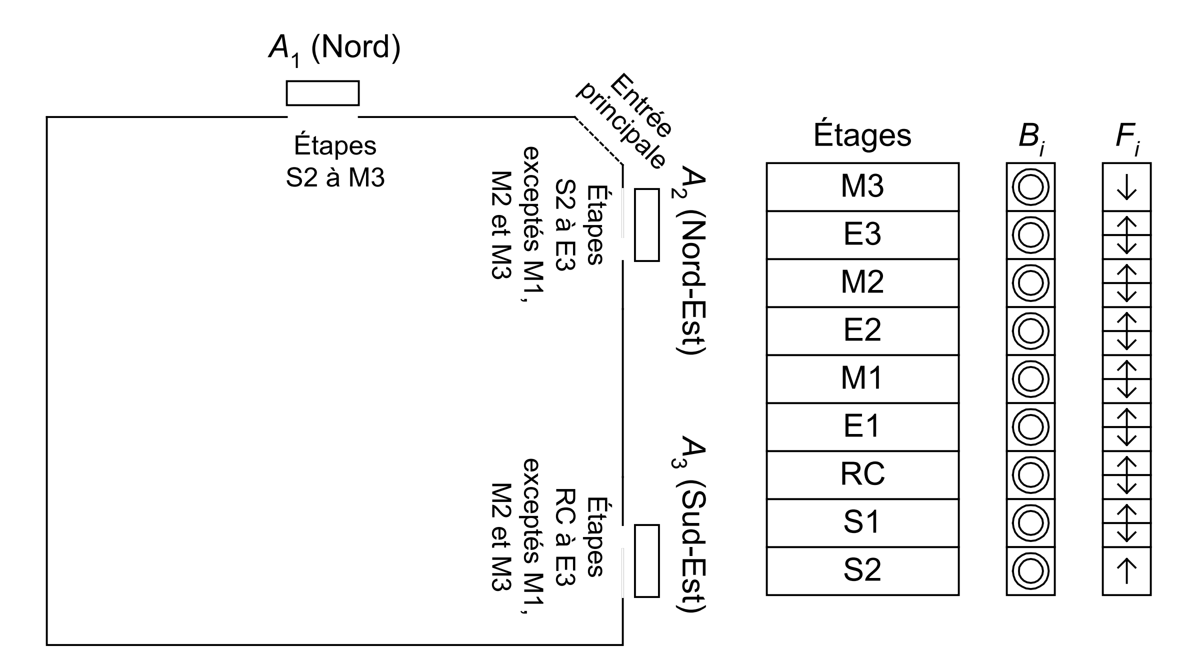


Figure 49 Emplacement des asscenseurs dans le camplus principal de l'ÉTS.

L'ascenseur A_1 parcourt les étages S2 à M3 inclusivement. L'ascenseur A_2 parcourt les étages S2 à M3 excepté les étages M1, M2 et M3. Finalement l'ascenseur A_3 parcoure les étages RC à E3 exceptés les étages mezzanines. Le travail consiste à réaliser un logiciel capable de simuler la logique de déplacement et le contrôle de ces ascenseurs.

Contraintes

Les boutons B_i sont situés à l'intérieur des ascenseurs. Les boutons F_i sont situés à l'extérieur des ascenseurs. ☞

- Chacun des ascenseurs dispose d'un ensemble de boutons B_i avec lumière témoin, un par étage déservi. La lumière s'allume lorsque le bouton correspondant est enfoncé. La lumière s'éteint lorsque l'ascenseur arrive à l'étage correspondant.

- Deux boutons (Haut – Bas) F_i avec lumière témoin sont disposés sur un panneau à l'extérieur des ascenseurs sur chacun des étages du campus. Pour des raisons évidentes, le premier étage et le dernier étage ne disposent qu'un seul bouton. La lumière de la direction désirée s'allume lorsque le bouton correspondant est enfoncé. Les lumières témoins éteignent (Haut – BAS) lorsque l'ascenseur arrive à l'étage de destination.

- Un ascenseur sans requête (libre) demeure sur l'étage courant avec les portes fermées.

- Un ascenseur doit toujours traiter d'abord les requêtes les plus près de sa position courante et dans la même direction courante. D'une manière plus précise, les règles de déplacement sont :

  - S'il n'y a pas de requête, un ascenseur doit demeurer sur son étage courant avec les portes fermées.

  - Un ascenseur doit traiter d'abord, traiter les requêtes venant des étages les plus près de sa position courante.

    L'ascenseur déplaçant vers le haut : traiter les requêtes vers le haut venant des étages supérieurs à la position actuelle de l'ascenseur. S'il n'y a plus requête de cette nature, traiter les requêtes pour la direction opposée. S'il n'y a aucune autre requête, l'ascenseur doit demeurer sur son étage courant avec les portes fermées.

    L'ascenseur déplaçant vers le bas : traiter les requêtes vers le bas venant des étages inférieurs à la position actuelle de l'ascenseur. S'il n'y a plus requête de cette nature, traiter les requêtes pour la direction opposée. S'il n'y a aucune autre requête, l'ascenseur doit demeurer sur son étage courant avec les portes fermées.

  - Une fois rendu à une destination, l'ascenseur doit ouvrir ses portes. Les portes doivent demeurer ouvertes pour une durée de temps fixe.

- Il est nécessaire de comptabiliser les statistiques d'utilisation des ascenseurs.

- Enfin, l'ascenseur se déplace à une vitesse constante. Par exempls Td est le temps nécessaire pour parcourir un étage. Pour atteindre n étages, le temps total est alors $n \times Td$.

4.8.1 ANALYSE PRÉLIMINAIRE

Dans notre contexte, les ascenseurs ont des états qui changent dans le temps (ils sont dynamiques). Ces états sont : *i*) l'étage courant de l'ascenseur; *ii*) le statut de l'ascenseur. L'étage courant est l'ensemble fini {S2, S1, RC, E1, M1, E2, M2, E3, M3}. Le fonctionnement de l'ascenseur est décrit par l'ensemble des valeurs de statut {REPOS, HAUT, BAS, ARRÊT}. La description de ces valeurs est donnée ci-dessous :

REPOS Portes fermées, immobile

HAUT Portes fermées, déplacement vers le haut
BAS Portes fermées, déplacement vers le bas
ARRÊT Portes ouvertes, immobile

D'un point de vue d'utilisation normale, une requête est exécutée de la manière suivante :

- Un passager, situé sur un étage, appuie sur le bouton F_i de l'étage en indiquant la direction désirée et la requête est enregistrée dans un tableau.

OU

- Un passenger, prenant place dans un ascenseur, appuie sur le bouton B_i du panneau de l'ascenseur et la requête est enregistrée dans un tableau.

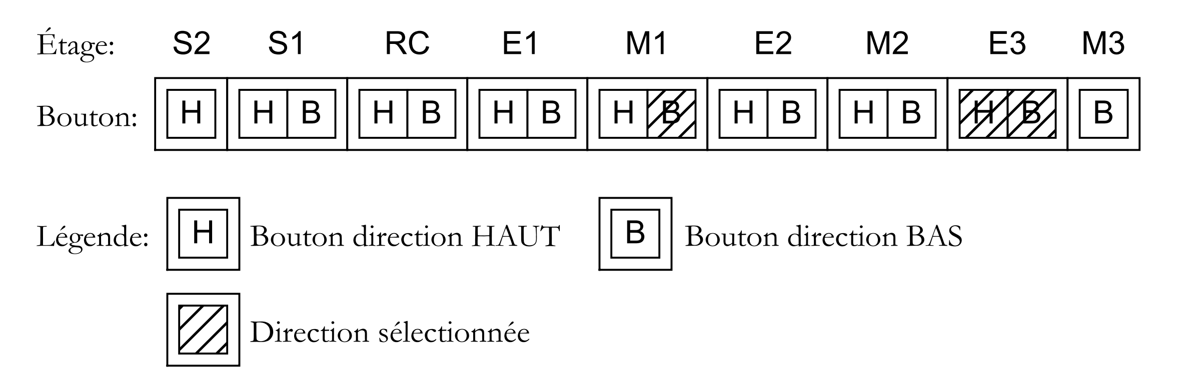


Figure 50 Disposition des boutons de l'ascenseur A1.

Ainsi, les boutons B_i et F_i peuvent être représentés par un tableau semblable à celui de la Figure 50. Les directions HAUT et BAS de chacun des étages sont indiquées. La direction sélectionée est indiquée par un carré hachuré. Note : Chaque ascenseur possède son propre tableau.

Les statistiques intéressantes à comptabiliser sont :

- Le nombre de requêtes effectuées par les passagers du système d'ascenseurs.
- Le nombre de changements de direction effectués par les ascenseurs.
- Le nombre d'arrêts effecutés par les ascenseurs.
- Le nombre de repos effectués par les ascenseurs.
- Le nombre d'étages parcourus par les ascenseurs.

Deux statuts de fonctionnement permettent à un ascenseur de choisir son parcours en appliquant les règles de déplacement présentées plus haut :

- Lorsque l'ascenseur est au REPOS.
- Lorsque l'ascenseur est à l'ARRÊT.

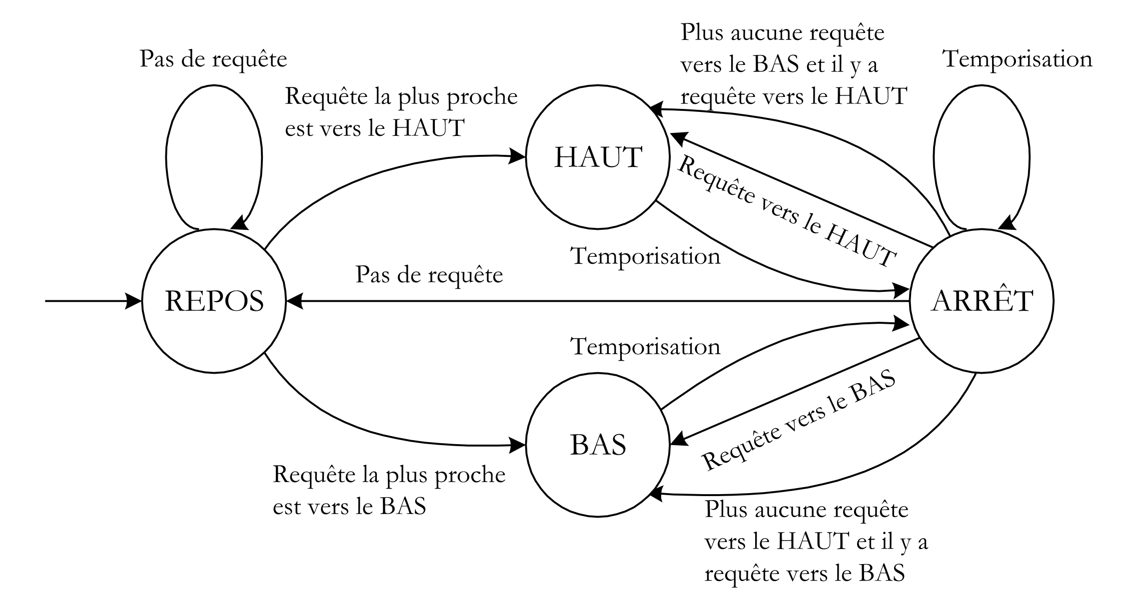


Figure 51 Diagramme d'état des ascenseurs.

Ce fait est exprimé dans la Figure 51. Lorsque l'ascenseur est à l'état REPOS, il peut passer à l'état HAUT ou à l'état BAS. La transition réelle effectuée dépendra de la nature des requêtes. Après une certaine temporisation (c'est-à-dire, le temps nécessaire pour atteindre la destination), l'ascenseur est à l'état ARRÊT. Cet état permet aux passager d'entrée et de sortir de l'ascenseur. L'ascenseur demeure dans cet état durant un temps constant. C'est également dans l'état ARRÊT que le système doit décider du prochain état de l'ascenseur : *i*) parcourir vers le haut; *ii*) parcourir vers le bas; *iii*) passer à l'état REPOS. À noter l'état REPOS est nécessairement précédé de l'état ARRÊT. Autrement dit, il n'est pas possible pour un ascenseur de s'immobiliser avec les portes fermées pendant son parcours (vers le haut ou vers le bas).

L'embarquement et débarquement des passagers d'un ascenseur sont réalisés de la manière suivante :

1. L'ascenseur se déplace jusqu'à l'étage de destination choisi par l'ascenseur. L'ascenseur dans le fonctionnement ARRÊT.

2. Les passagers entrent et/ou sortent de l'ascenseur. Les passagers peuvent sélectionner un bouton B_i. en tout moment. Il en est de même pour les utilisateurs sur les étages (pour les boutons F_i).

3. Après un temps constant, les portes de l'ascenseur se referment.

4. Recommencer l'étape 1.

4.8.2 QUELQUES DIAGRAMMES UML

Voici quelques diagrammes UML dégagés lors de l'analyse préliminaire effectuée précédemment. La nature de ces diagrammes ainsi que les techniques utilisées dans l'obtention de ces diagrammes seront discutées dans les chapitres subséquents de ce document.

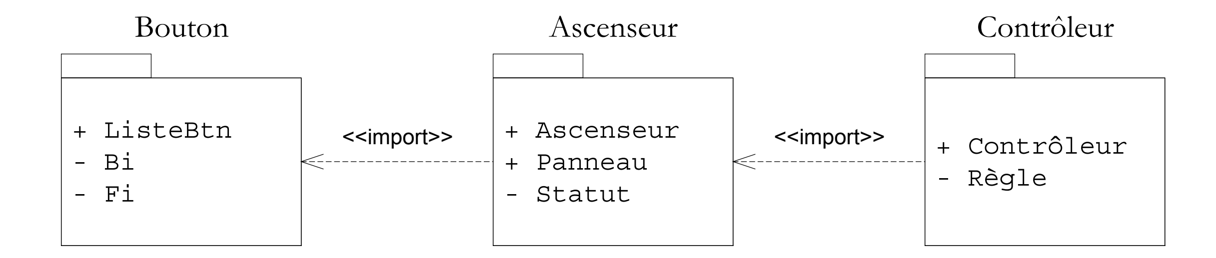


Figure 52 Organisation des modèles en paquets.

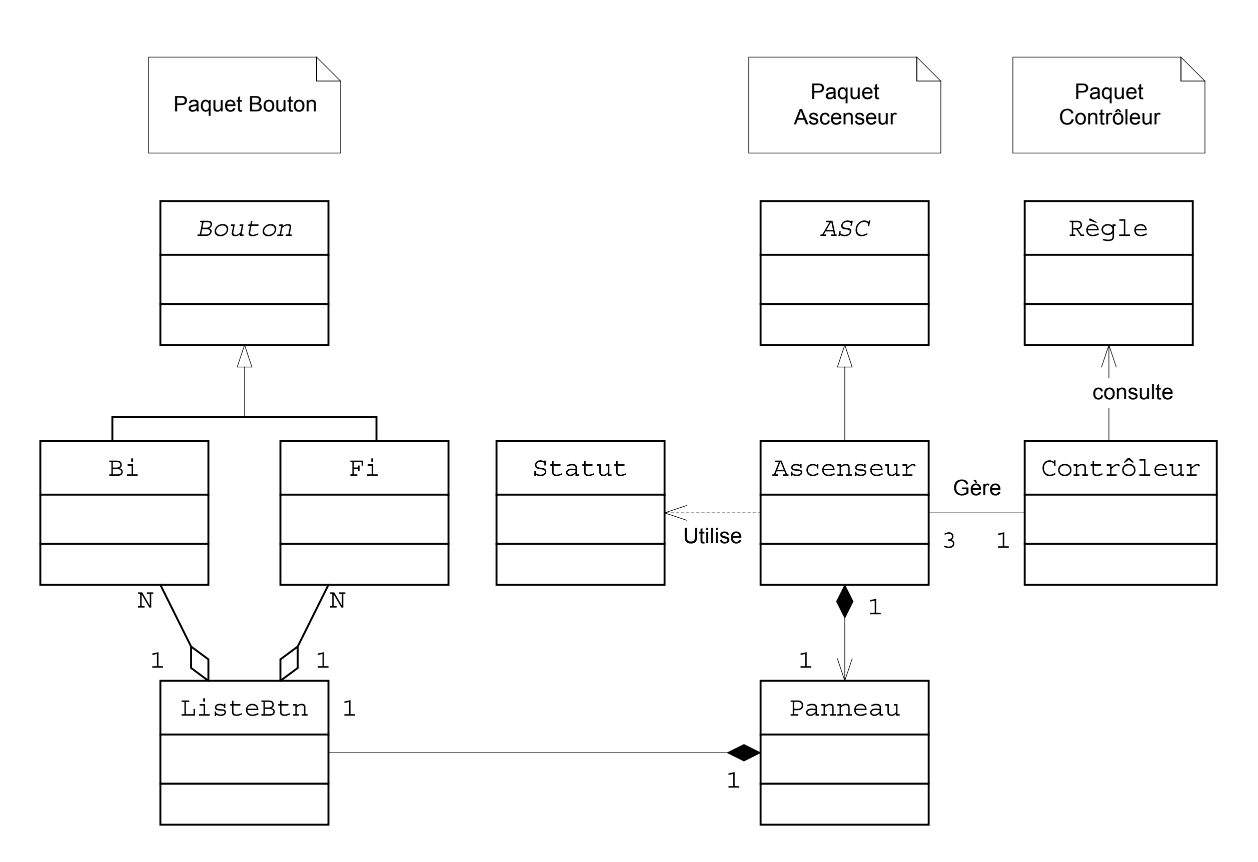


Figure 53 Diagramme de classes du système d'ascenseur.

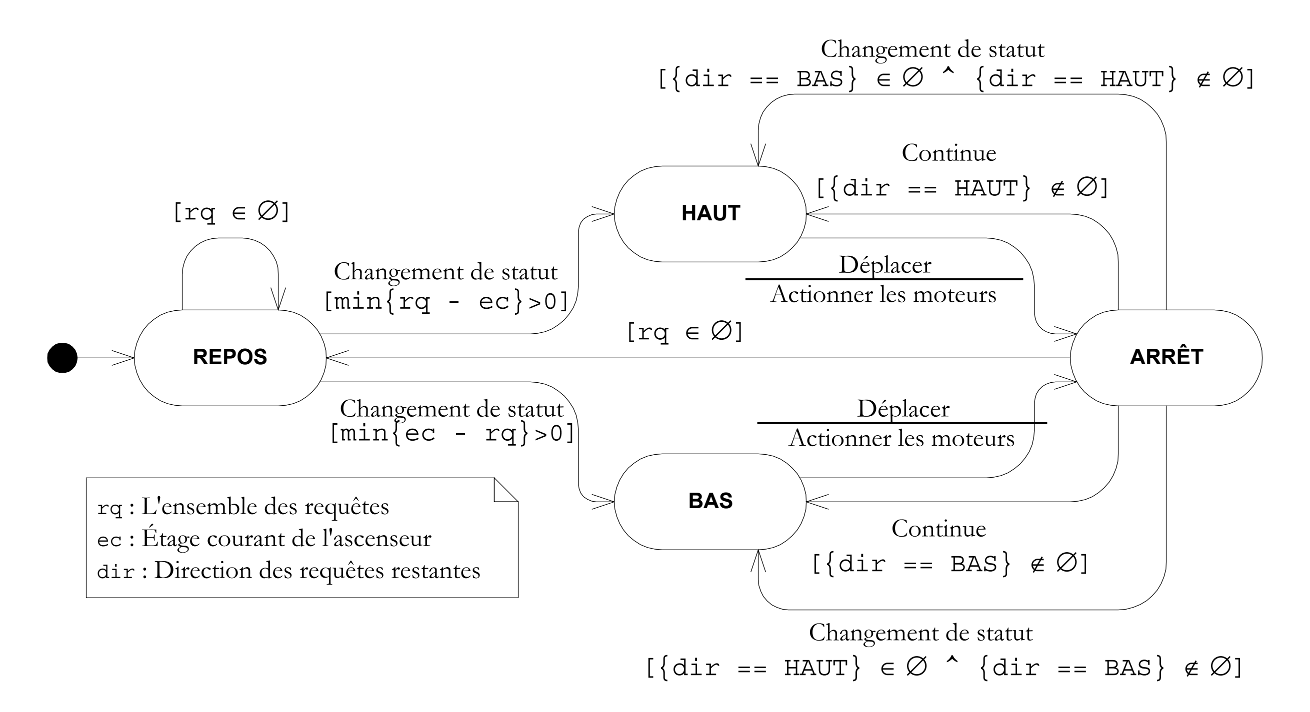


Figure 54 Diagramme d'état des ascenseurs.

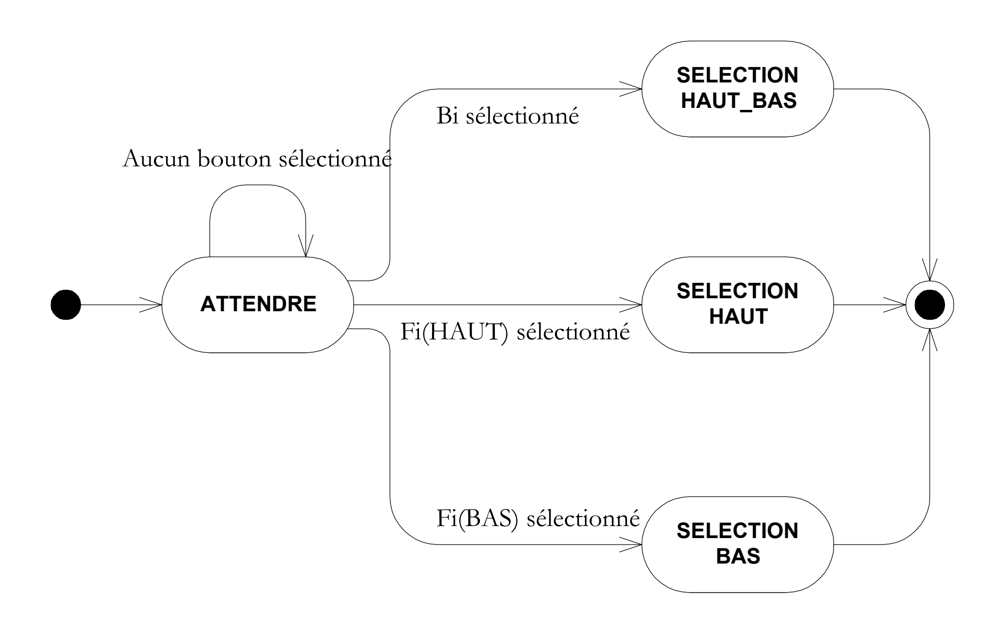


Figure 55 Diagramme d'état des panneaux de boutons.

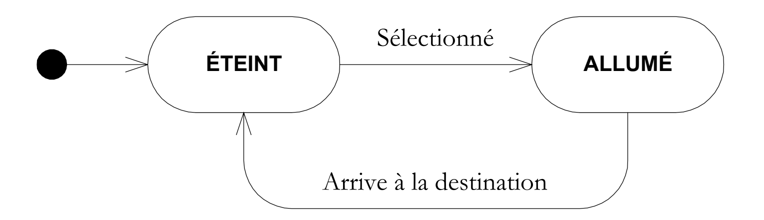


Figure 56 Diagramme d'état des boutons.

LECTURE SUGGÉRÉE

Ce chapitre est une présentation générale de l'approche orientée objet. Nous avons passé en revue les notions de l'objet et de classe dans le contexte du génie logiciel.

Les références ci-dessous renferment toutes les informations utiles pour la compréhension du sujet de l'approche orientée objet. Ces références ont été utilisées pour la production de ce chapitre.

[MULL97] Muller, Pierre-Alain, *Instant UML*, Wrox Press, 1997.

Ce livre traite d'une manière pratique les éléments de l'approche orientée objet et surtout la notation UML.

[SOMM95] Sommerville, Ian, *Software Engineering*, Addison-Wesley, 1995.

Le livre de Sommerville aborde le sujet du génie logiciel. Il explique le rôle de l'approche orientée objet dans le contexte général du développement logiciel.

[LIBE98] Liberty, Jesse, *Beginning Object-Oriented Analysis and Design*, Wrox Press, 1998.

Cette dernière référence s'adresse aux programmeurs C++ qui désirent mieux connaître les techniques d'analyse et de conception orientées objet. À noter que la présentation de Jesse Liberty est surtout concernée par les aspects application des techniques.

[HTTP01] http://www.geocities.com/SiliconValley/Network/1582/uml-example.html/

L'exemple de la section 4.8 est inspiré de l'exemple présenté sur cette page.

PROBLÈMES

** | 4.1 Un objet est défini par la formule *Objet = Attribut + Opération + Identité*. Dégager les attributs, les opérations et l'identité d'un objet représentant les étudiants du cours GPA789.

*** | 4.2 Exprimer par la notation UML la situation où trois ordinateurs utilisent une seule et même imprimante.

* | 4.3 Quelle est la différence fondamentale entre une agrégation et une composition.

*** | 4.4 Décrire par la notation UML les relations impliquant un conducteur, le moteur, les pneus ainsi la plaque d'immatriculation d'une voiture.

*** | 4.5 Pour le laboratoire #1, refaire le diagramme des classes en utilisant la notation UML. présentée dans ce chapitre.

*** | 4.6 Pour le laboratoire #1, refaire le diagramme des objets en utilisant la notation UML et les notions présentées dans ce chapitre (c'est-à-dire, spécifier les relations et la multiplicité).

**** | 4.7 Suggérer un exemple dans lequel on peut vérifier le principe de substitution.

*** | 4.8 Donner le diagramme de collaboration des objets du laboratoire #1.

*** | 4.9 Donner le diagramme de squence des objets du laboratoire #1.

CHAPITRE 5

Notation UML

Un philosophe contemporain, argumentateur à outrance, auquel on représentait que ses raisonnements irréprochablement déduits avaient l'expérience contre eux, mit fin à la discussion par cette simple parole : « L'expérience a tort ».

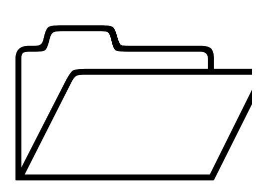
— *Henri Bergson, Le rire.*

La notation UML (*Unified Modeling Language*) n'est pas une formalisation du processus de développement orienté objet. Elle sert plutôt à décrire les éléments logiciels obtenus par l'application des techniques de développement. La notation UML comprend neuf (9) diagrammes pour représenter différents points de vue de la modélisation. L'utilisation de UML est très flexible puisqu'elle est indépendante des langages de programmation. Elle est également une notation extensible. Nous pouvons représenter un concept particulier en utilisant les mécanismes d'extension prévus.

5. NOTATION UML

Dans la version 1.1 de UML il existe neuf (9) types de diagramme. Ils sont présentés ci-dessous. Ces diagrammes UML seront présentés dans les sous-sections suivantes.

- **Diagramme de classes** : Description statique de la structure en termes de classes et relations.
- **Diagramme d'objets** : Représente les objets et leurs relations.
- **Diagramme d'activités** : Représente le comportement d'une opération sous la forme d'un ensemble d'actions.
- **Diagramme de séquence** : Représente les objets, les liens et les interactions où la dimension temporelle est prédominante.



- **Diagramme de collaboration** : Représente les objets, les liens et les interactions où la dimension spatiale est prédominante.
- **Diagramme des composants** : Représente les composants d'une application.

- **Diagramme de déploiement** : Représente l'application des composants sur le matériel.

- **Diagramme d'état** : Représente le comportement d'une classe en termes d'état.

- **Diagramme de cas d'utilisation** : Représente l'aspect fonctionnel d'une application d'un point de vue utilisateur.

La notation UML est composée d'éléments de base. Ces éléments de base sont : *i*) éléments de modélisation; *ii*) éléments visuels. Les éléments de modélisation représentent l'abstraction de l'application tandis que les éléments visuels aide à faciliter la manipulation des éléments de modélisation. Aussi, les éléments sont presque toujours regroupés dans des paquets (*packages*). Dans le UML, un modèle est, une abstraction d'une application, représenté par une hiérarchie d'ensembles. Voici le diagramme UML montrant les éléments d'un modèle (au sens UML du terme).

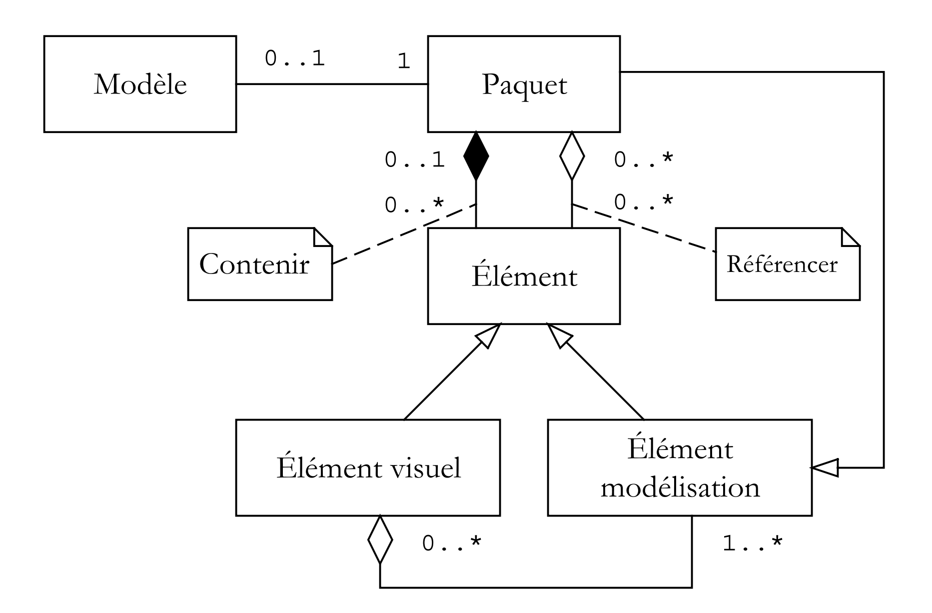


Figure 57 Éléments UML d'un modèle.

Un paquet peut donc représenter zéro (0) ou un (1) modèle. Il peut contenir de zéro (0) à un nombre quelconque d'éléments. Un paquet peut également référencer zéro (0) à un nombre quelconque d'éléments. Un élément peut être un élément visuel ou un élément de modélisation. Enfin, les éléments visuels servent à représenter les éléments de modélisation.

5.1 MÉCAMISMES COMMUNS UML

Le UML est une notation **consistante**. C'est-à-dire, il n'existe pas de contradiction interne dans la symbologie utilisée. L'intégrité des concepts projetés par la notation UML est assurée grâce aux mécanismes communs définis dans le langage. Ces mécanismes sont :

- **Stéréotypes** : utilisés lorsque la sémantique (le sens) des éléments de base est insuffisante pour exprimer la notion véhiculée. Il est alors possible d'ajouter un ou plusieurs stéréotypes pour étendre la gamme d'éléments de modélisation disponible. Les stéréotypes sont également utilisés pour unir des concepts reliés.

- **Valeurs marquées** : sont sous forme de couples (nom, valeur). Elles décrivent des propriétés d'un élément de modélisation. Une valeur marquée modifie la sémantique de l'élément relié.

- **Notes** : commentaires attachés aux éléments de modélisation. Une note ne possède pas de signification particulière et ne contribue pas à la sémantique du modèle. Cependant, une note peut devenir une contrainte par l'application d'un stéréotype.

- **Contraintes** : jouent le rôle d'une relation sémantique entre les éléments de modélisation. La syntaxe des contraintes est libre. Cependant, le texte des contraintes est toujours placé entre accolades.

- **Dépendances** : définies une relation unidirectionnelle entre deux éléments de modélisation. Une relation de dépendance implique toujours un élément de modélisation appelé source et un élément de modélisation appelé cible. Les notes et les contraintes peuvent jouer le rôle de source dans une relation de dépendance.

- **Type/Instance** : un type est l'essence de l'élément alors qu'une instance est la manifestation de ce type.

- **Type/Classe** : dans ce cas, un type est la spécification d'un élément alors que la classe est l'implantation de cette spécification.

5.2 TYPES DE DONNÉES DU LANGAGE

Les types de données ne sont pas des éléments de modélisation. Ils ne possèdent pas de stéréotypes, valeurs marquées ou contraintes. Le donne les types de données UML.

Type	Signification
Booléen	N'admet que `TRUE` et `FALSE`.
Expression	Chaîne de caractères.
Multiplicité	Ensemble non nul d'entiers non négatifs.
Nom	Chaîne de caractère identifiant un élément.
Entier	Comprend les entiers positifs et négatifs.
Chaîne	Séquence de caractères.
Temps	Chaîne de caractères représentant le temps relatif ou absolu.
Non interprété	Type dont la sémantique n'est pas définie dans le UML.

Tableau 15 Type de base dans le UML.

5.3 PAQUETS UML

Un paquet (*package*) UML est utilisé pour regrouper les éléments de modélisation. Le symbole de dossier suivant représente un paquet UML :

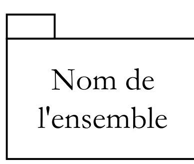


Figure 58 Symbole UML représentant un paquet (package).

Nous pouvons considérer les paquets UML comme des dossiers d'un système de classement. Chaque paquet est un sous-ensemble d'un modèle comportant des classes, des objets, des relations et leurs diagrammes associés. L'ensemble des paquets forme le modèle complet de l'application en développement.

À noter que la séparation d'un modèle en paquets UML **n'est pas** une décomposition fonctionnelle. En effet, la séparation d'un modèle en paquets est un processus intellectuel qui doit être basé uniquement sur des critères logiques. Ainsi, la hiérarchie des paquets et leurs interdépendances forment ce que l'on appelle l'**architecture du système** selon l'approche orientée objet.

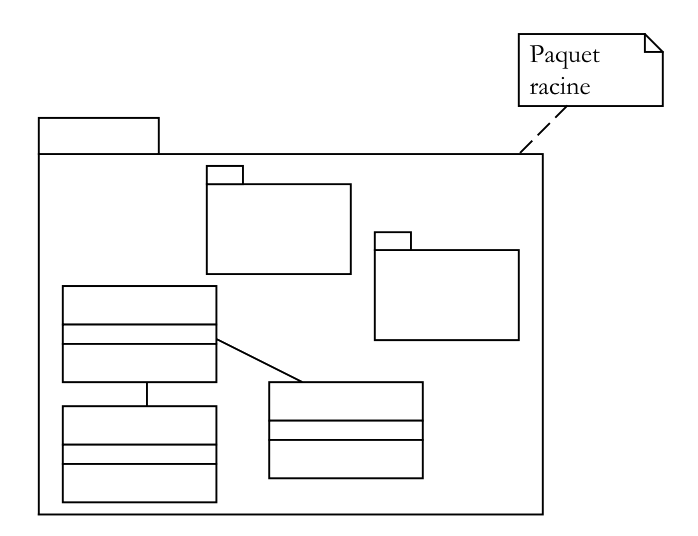
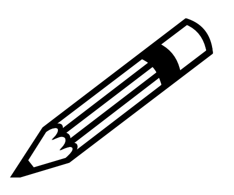


Figure 59 Organisation d'un modèle en paquets UML. Le paquet racine est le dossier initial de ce système de classement.



Tout comme le mot de clé `namespace` en C++, un paquet UML défini implicitement un espace de nom qui lui est propre. Il est donc possible de déclarer deux éléments de même nom pourvu qu'ils sont contenus dans deux paquets différents. Aussi, les paquets peuvent s'imbriquer les uns dans les autres (un dossier peut contenir d'autres dossiers). La Figure 59 montre l'organisation possible des paquets UML.

Tout comme la directive de pré-compilation `#include` de C++, une classe contenue dans un paquet peut être importée dans un autre paquet. L'importation des éléments est réalisée par une relation stéréotypée « `import` ». La Figure 60 exprime l'utilisation des éléments du paquet B dans le paquet A à l'aide de la relation stéréotypée « `import` ». La déclaration d'une classe dans ce contexte aura la forme `NomDePaquet::NomDeClasse` où `NomDePaquet` représente le nom d'un paquet, le symbole `::` est l'opérateur de résolution et `NomDeClasse` est le nom d'une classe dans le paquet `NomDePaquet`.

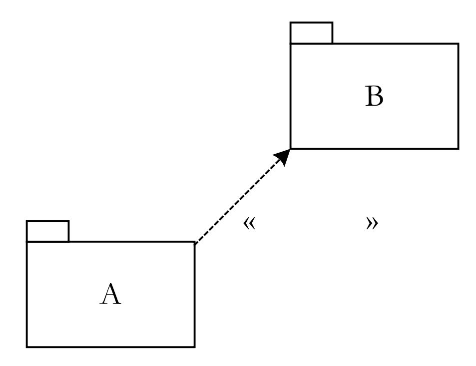


Figure 60 Paquet `A` importe les service de paquet `B`.

Les classes d'un paquet ne sont pas accessibles à l'extérieur du paquet inconditionnellement. En effet, chaque élément contenu dans un paquet est qualifié par un paramètre `public` ou `implementation`. Les classes étiquetées `public` sont visibles à l'extérieur du paquet alors que celles paramétrées `implementation` sont uniquement utilisables dans le paquet qui l'englobe.

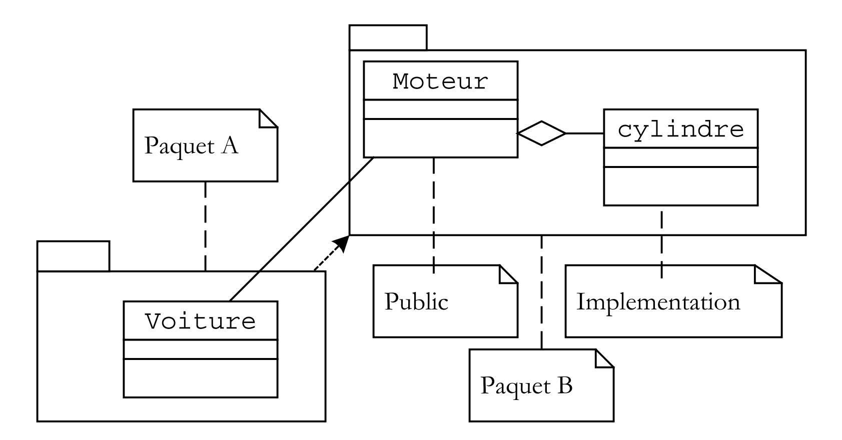
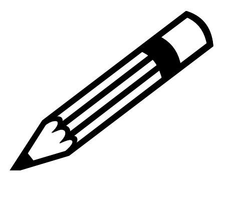


Figure 61 Importation des classes entre paquets est contrôlée par les paramètres `public` et `implementation`.

Reprenant l'exemple de la Figure 60, nous pouvons expliciter la relation stéréotypée « `import` » par celle de la Figure 61. Ainsi, la classe `Voiture` du paquet `A` est en relation (via un lien) avec la classe `Moteur` du paquet `B`. La classe `Moteur` est marquée publique. Elle est donc visible à l'extérieur du paquet `B`.



Enfin, il est fortement déconseillé de créer des paquets qui dépendent les uns des autres. Les relations croisées entre deux paquets et les relations circulaires impliquant plusieurs paquets sont à proscrire. Ces relations rendent l'implantation concrète des classes difficiles à réaliser et maintenir.

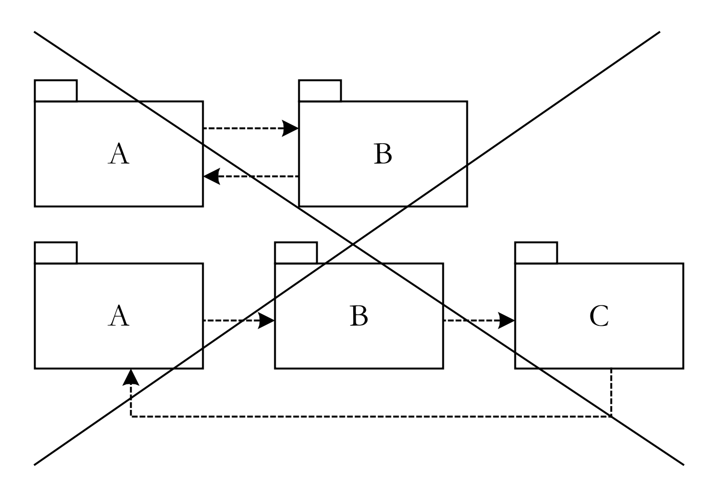


Figure 62 Ces types de relations sont à éviter.

Finalement, certains paquets UML sont d'usage général. Les relations de ces paquets ne sont pas toujours montrées dans un diagramme. Par exemple, la gestion des erreurs, les structures de données de base sont normalement des paquets à visibilité globale. Il n'est pas nécessaire de toujours montrer leurs interdépendances avec les autres paquets du modèle.

5.4 DIAGRAMMES DE CLASSE

Nous avons vu dans la section 4.7 (page 160) qu'une classe est la description statique des objets d'un domaine particulier d'application. La représentation UML d'une classe est un rectangle contenant trois compartiments.

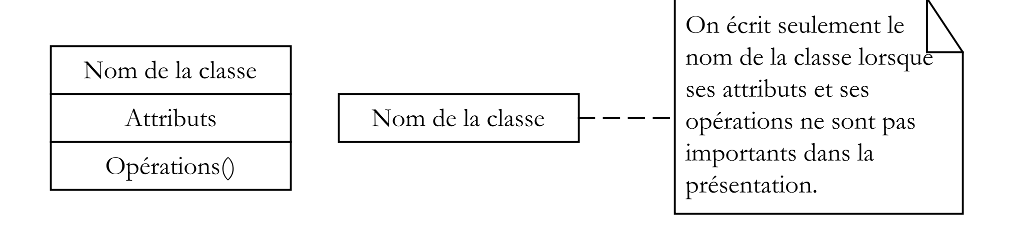


Figure 63 Symboles représentant une classe.

Le symbole d'une classe peut également contenir un stéréotype et des **propriétés**. Les stéréotypes appropriés pour une classe sont énumérés dans le Tableau 16. Les propriétés d'une classe englobent toutes les valeurs attachées à un élément de modélisation (c'est-à-dire, attributs, associations et les valeurs marquées).

Stéréotype de classe	Signification
« signal »	Événement qui enclenche un changement d'état.
« interface »	Description d'une opération visible à l'extérieur de la classe.
« metaclass »	Classe d'une classe.
« utility »	Classe considérée comme un module et ne peut être instanciée.

Tableau 16 Stéréotypes applicables à une classe.

Voici la symbologie UML qui est utilisable dans le compartiment désignant le nom d'une classe.

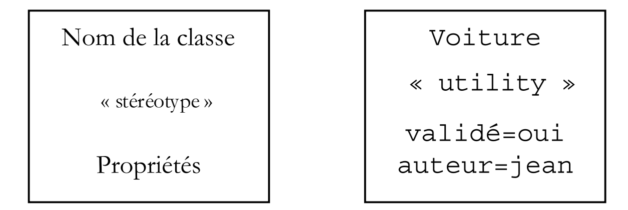


Figure 64 Contenu possible du compartiment réservé pour le nom de la classe.

5.4.1 ATTRIBUTS ET OPÉRATIONS

Il n'est pas toujours nécessaire d'énumérer tous les attributs et opérations dans les compartiments d'une classe. Par contre, les attributs et opérations qui sont présentés doivent être écrits selon le format suivant :

Attributs :

```
nom_attribut : type_attribut = valeur_initiale
```

Opérations :

```
nom_operation (nom_param : type_param = valeur_defaut, …) : type_retour
```

Point
CoordonnéesX : entier = 0 CoordonnéesY : entier = 0
Affichage() : booléen

Figure 65 Syntaxe UML des attributs et opérations.

Encore une fois, la notation UML est indépendante des langages de programmation. Ainsi, le type des attributs et opérations n'est pas celui de C++ ou Java.

5.4.2 CLASSES PARAMÉTRISÉES

Les classes paramétrisées sont des modèles de classe. Une classe paramétrisée doit être instanciée pour obtenir une classe typée. C'est à partir de la classe typée que l'on peut créer des objets. En C++, les classes de la bibliothèque STL sont des classes paramétrisées. Normalement, les classes paramétrisées sont définies lors de la phase de conception de l'application. Voici la notation UML utilisée pour représenter les classes paramétrisées.

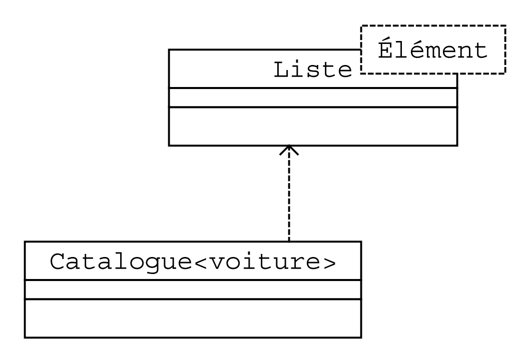


Figure 66 Notation UML pour les classes paramétrisées.

Dans la figure ci-dessus, La classe `Liste` est une classe paramétrisée. Le paramètre de cette classe est `Élément`. L'instanciation de cette classe paramétrisée donne une classe typée `Catalogue` dont le paramètre est `voiture`. La classe typée `Catalogue` est une instance de la classe paramétrisée `Liste` et elles sont reliées par une ligne fléchée en pointillé. Nous pouvons créer des objets utilisables à partir de la classe typée `Catalogue`.

5.4.3 CLASSES UTILITAIRES

Les classes utilitaires (*utility classes*) ne sont pas des classes dans le sens de l'approche orientée objet. Elles servent plutôt à représenter des modules regroupant des fonctions semblables. Par exemple, il est possible de regrouper dans un module les fonctions nécessaires pour réaliser l'inversion des matrices creuses. Ce module n'est pas une classe proprement dite mais le UML permet de le représenter par une classe utilitaire. Enfin, on ne peut pas créer un objet à partir d'une classe utilitaire.

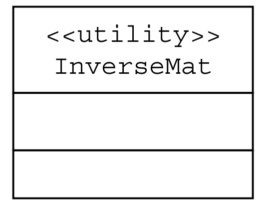


Figure 67 Représentation d'une classe utilitaire.

5.5 Relations entre les classes

Dans le langage UML, il existe trois relations principales entre les classes. Elles sont : *i*) association; *ii*) généralisation; *iii*) dépendance. Ces relations ainsi que leur représentation graphique sont l'objet d'étude de cette sous-section.

5.5.1 Association

Nous avons vu au chapitre 4 (section 4.7.3) qu'une association est une relation structurelle entre les classes. Pour simplifier la compréhension, nous devons donner un nom à l'association, un rôle aux classes impliquées et un nombre aux objets associés. Il est également recommandé d'utiliser des verbes pour désigner le type d'association entre les classes.

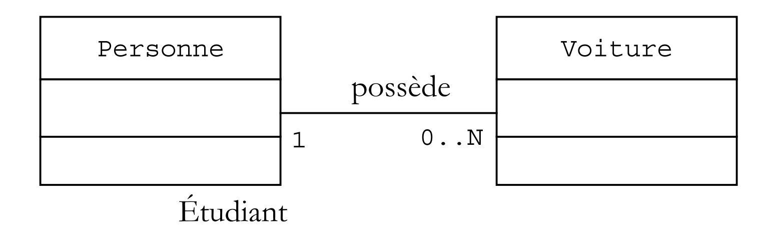


Figure 68 Notation UML d'une association.

Dans la Figure 68, il existe une relation de possession entre la classe `Personne` et la classe `Voiture`. À cause du rôle donné à la classe `Personne` et la multiplicité nous indiquent qu'il s'agit d'un étudiant qui peut posséder 0 à N voiture. La multiplicité des associations indique le nombre maximal d'objets de chacune des classes que l'on prévoit instancier. Il est clair que l'information de la multiplicité ne donne pas l'ordre de création des objets.

5.5.2 Contraintes des associations

Les contraintes sont des spécifications appliquées à des liens entre classes. Dans le cas de la multiplicité, il s'agit d'une contrainte donnée sur le nombre d'objets associés aux classes. D'autres contraintes sont applicables à des associations de classes. La syntaxe des contraintes est simple, il suffit de les placer entre accolades :

`{ordered}` sert à indiquer une relation d'ordre dans une collection. Une collection est un terme générique désignant tout ensemble d'objet dont la cardinalité est supérieure à 1. Par exemple, un étudiant peut posséder plus d'une voiture. Dans ce cas, nous pouvons spécifier la collection de voiture par la contrainte `{ordered}`. Cette contrainte ne dicte pas la mécanique de l'ordonnancement, elle spécifie seulement qu'il existe une relation d'ordre parmi les objets de la collection.

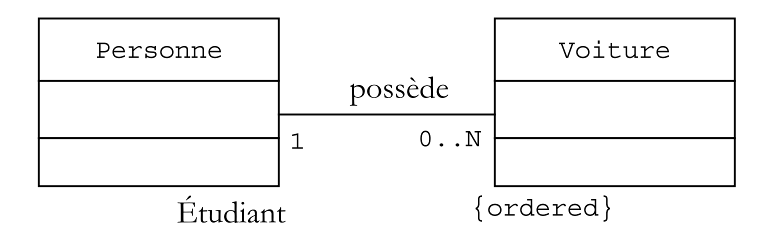
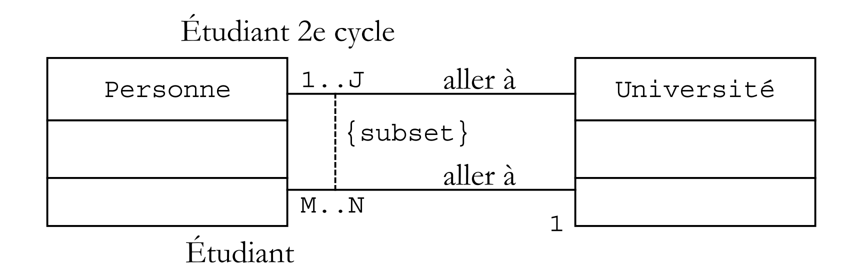
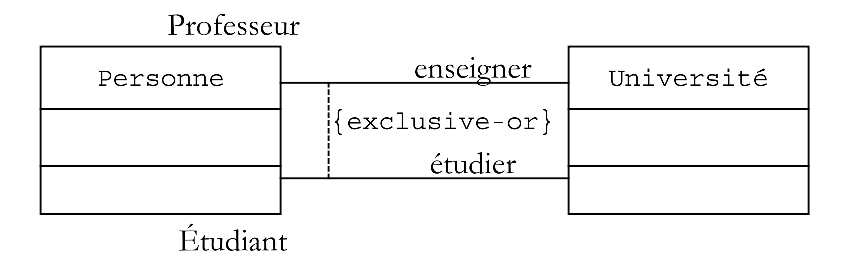


Figure 69 Collection d'objets ordonnés spécifiée par la contrainte `{ordered}`.

`{subset}` indique une collection est contenue dans une autre collection (sous-ensemble). Par exemple, parmi l'ensemble des étudiants, il existe un sous-ensemble qui sont des étudiants de 2[e] cycle. La notation UML utilisée pour indiquer ce fait est une ligne pointillée reliant les rôles et la constrainte `{subset}` près de cette ligne pointillée.

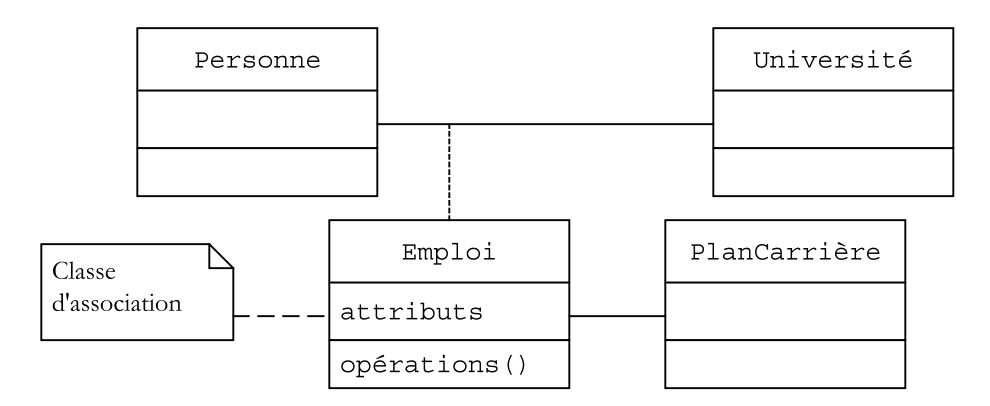


`{exclusive-or}` est une contrainte qui signifie qu'une seule association est valide parmi un ensemble d'associations. Ainsi, selon la constrainte `{exclusive-or}` ci-dessous la classe `Personne` joue le rôle de professeur ou le rôle d'étudiant. Mais seulement **l'une** des deux associations est valide.

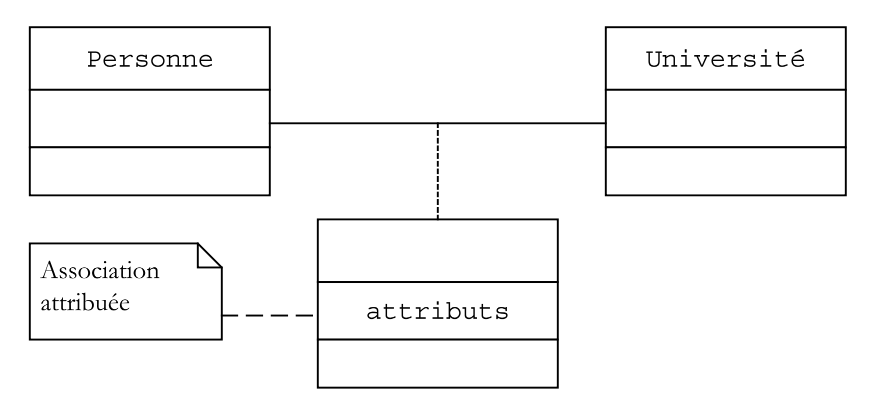


5.5.3 CLASSES D'ASSOCIATION

Il est possible de représenter une association par une classe. Cette façon de représenter une association permet l'ajout des attributs et opérations à une association. Par rapport à une association ordinaire, une classe d'association donne une plus grande flexibilité de représentation. Pour une classe d'association, on la relie par une ligne pointillée vers le lien concerné. Normalement une classe d'association peut aussi entrer en relation avec d'autres classes.

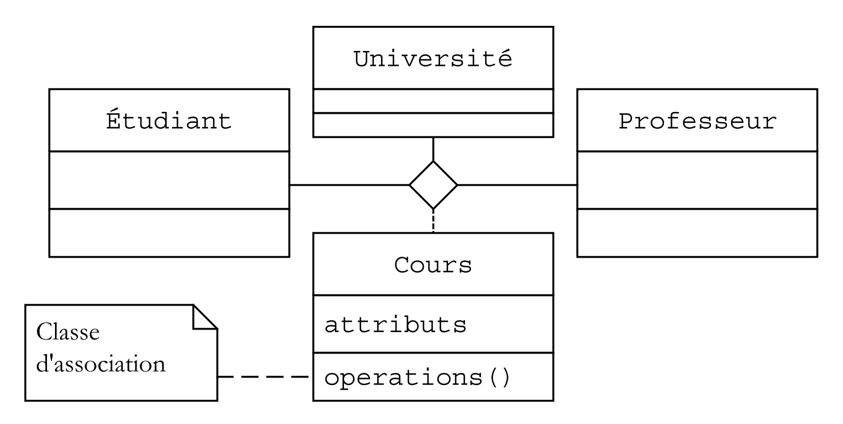


Une classe d'association qui n'a que des attributs est appelée une **association attribuée**. Une classe de ce type n'entre pas en relation avec d'autres classes et n'a pas de nom.



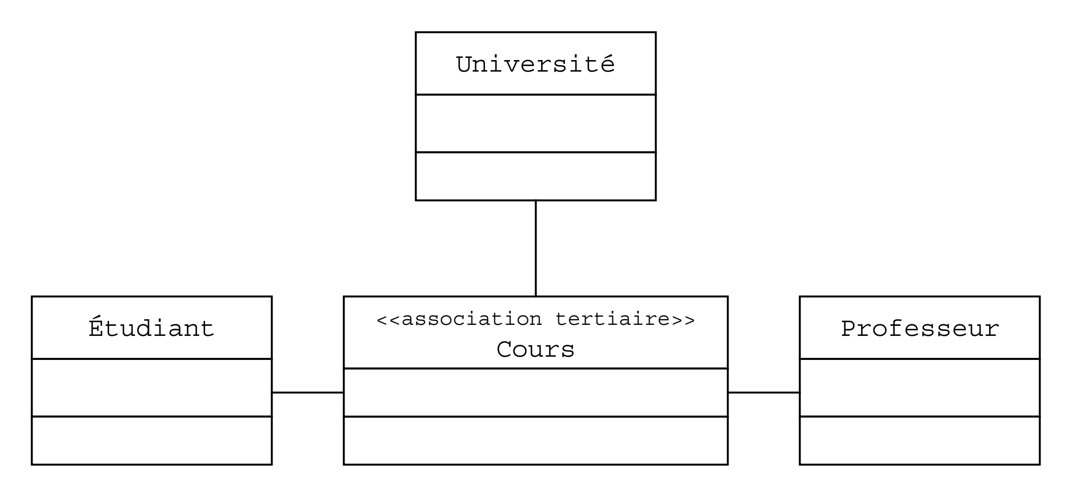
5.5.4 ASSOCIATIONS N-AIRE

La plupart des associations présentées jusqu'à présent sont des associations binaires. C'est-à-dire, des associations reliant deux classes. Pour représenter des associations reliant N classes, nous utilisons la symbologie suivante :



Dans cet exemple nous avons utilisé une association tertiaire pour indiquer les classes impliquées dans un cours universitaire. La classe d'association `Cours` montre clairement la signification de cette association.

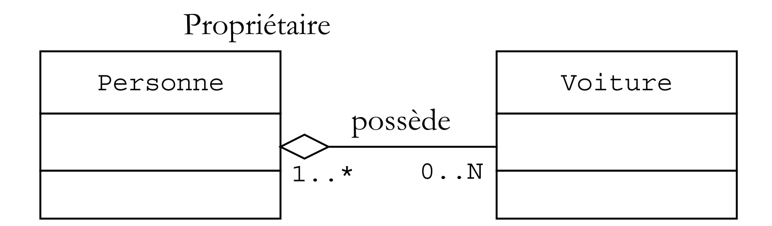
Nous pouvons éviter l'utilisation de la symbologie ◇ en faisant une promotion de la classe d'association `Cours`. Ainsi, la classe Cours peut devenir une véritable classe ayant des liens avec les classes `Professeur`, `Étudiant` et `Université`. La classe `Cours` est alors dotée d'un stéréotype indiquant une relation tertiaire.



5.5.5 AGRÉGATIONS

Une agrégation représente une relation asymétrique dans laquelle l'une des classes impliquées est plus importante que les autres. Pour déterminer une relation d'agrégation, appliquer les critères suivants :

- Une classe fait partie d'une autre classe.
- Les valeurs d'attribut d'une classe sont propagées dans les valeurs d'attribut d'une autre classe.
- Une action appliquée à une classe implique une action sur une autre classe.
- Les objets d'une classe sont subordonnés aux objets d'une autre classe.

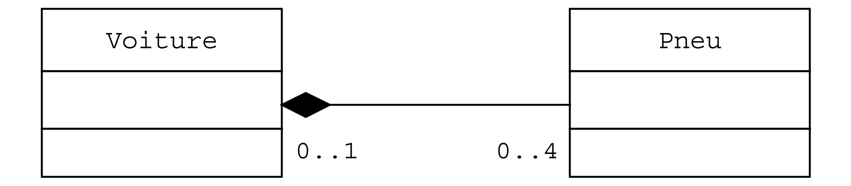


Dans l'exemple ci-dessus la relation d'agrégation est utilisée pour indiquer le concept de propriété multiple. Ainsi, il est possible que deux personnes ou plus soient

propriétaires de 0 à N voitures. Les objets de la classe `Voiture` sont contrôlés par les objets de la classe `Personne` puisque ces derniers sont les propriétaires. Normalement, les propriétaires peuvent conduire, vendre et même détruire les voitures qu'ils possèdent.

5.5.6 COMPOSITION

La composition est une forme particulière d'agrégation. La composition signifie qu'il existe un confinement physique des classes les unes dans les autres. La multiplicité de la composition est toujours 0 à 1.



Cette notation UML signifie que la classe `Voiture` contient physiquement 0 à 4 instances (objets) de la classe `Pneu`.

5.5.7 GÉNÉRALISATION

Dans le UML, la génération est un concept appliqué à une hiérarchie de classes, à des paquets et à des cas d'utilisation. La généralisation représente une relation de type « est-un ». Ainsi, une voiture est un véhicule, un chat est un animal, etc. Le concept de généralisation est beaucoup abstrait que le concept d'héritage des langages de programmation orientés objet. En fait, l'héritage est une méthode d'implémentation de généralisation. Il existe également plusieurs contraintes possibles dans une relation de généralisation.

`{exlcusive}` **Contrainte par défaut de la généralisation**. Elle signifie qu'un objet est une instance d'une des sous-classes de la hiérarchie.

`{disjoint}` Contrainte indiquant qu'une classe descendante de classe `C` (par exemple) est nécessairement une descendente d'une des sous-classes de `C`. Autrement dit, la contrainte `{disjoint}` limite les sous-classes à un seul parent.

`{overlapping}` Contrainte indiquant qu'une classe descendante de classe `C` (par exemple) appartienne au produit cartésien des sous-classes de C. Autrement dit, la contrainte `{overlapping}` indique qu'une classe possède un parentage multiple.

`{complete}` Indique que la généralisation est complète. Il n'est plus possible d'ajouter de nouvelles sous-classes dans la hiérarchie.

`{incomplete}` Indique que la généralisation est incomplète. Il est possible d'ajouter de nouvelles sous-classes dans la hiérarchie.

Par exemple, la figure ci-dessous donne une hiérarchie de classes dans laquelle la contrainte {incomplete} est appliquée. On peut appliquer les autres contraintes de la même façon.

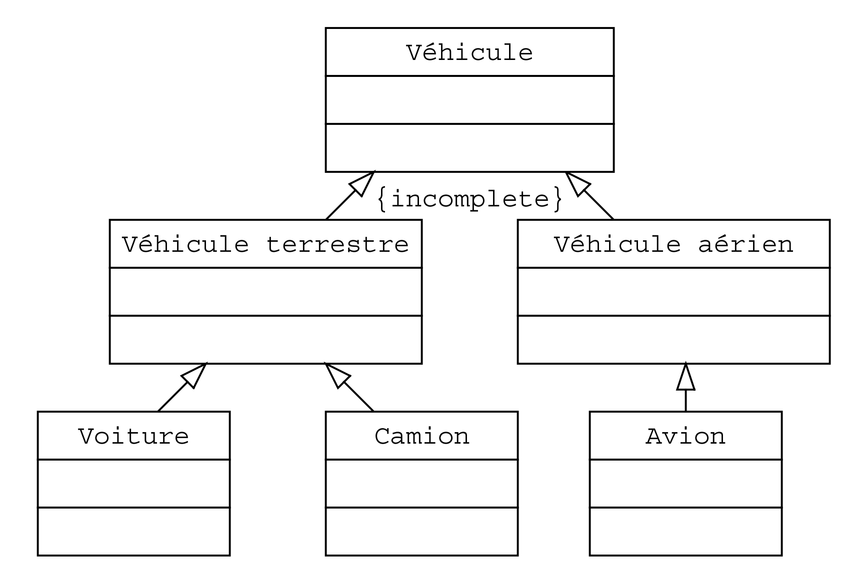
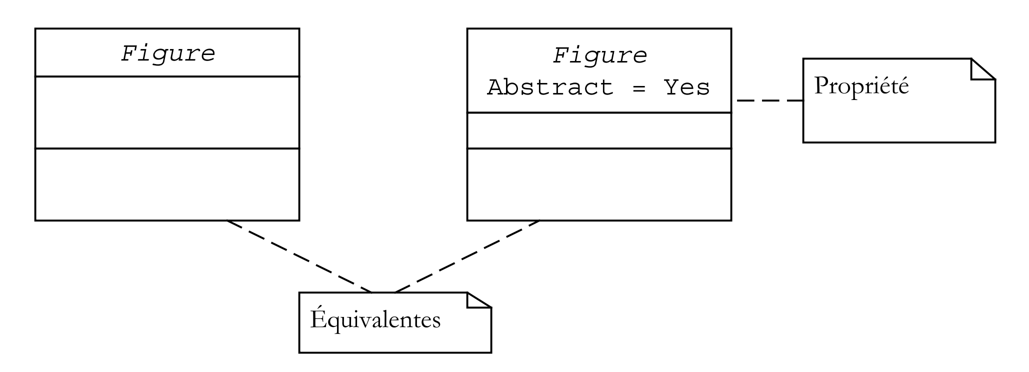


Figure 70 Hiérarchie de classes incomplète.

5.5.8 CLASSES ABSTRAITES

Une classe abstraite ne peut donner des objets. Elle sert plutôt à produire une spécification générale qui sera la base de d'autres classes. Nous décrivons dans les classes abstraites les mécanismes généraux tout en mettant de côté les capacités particulières des classes concrètes. Les classes abstraites sont fort utiles pour créer des applications logicielles extensibles. En effet, les nouveaux besoins, les extensions et améliorations sont implantées dans les nouvelles sous-classes.

Une classe abstraite possède la propriété `Abstract` réglée à vraie. Aussi par convention, on écrit le nom d'une classe abstraite en italique. La propriété `Abstract` est également applicable aux opérations. Dans ce cas, les opérations abstraites signifient que le corps de l'opération doit être défini dans les sous-classes. Enfin par souci de simplicité il est permis d'omettre la propriété `Abstract` dans la représentation graphique des classes abstraites.



5.6 DIAGRAMMES DE CAS D'UTILISATION

Les cas d'utilisation (*use cases*) servent à décrire le comportement d'une application selon d'un point de vue d'utilisateur. Ils établissent donc les relations entre l'application et son environnement d'utilisation. En général, les cas d'utilisation représentent les fonctionnalités d'une application en illustrant les réponses de l'application face aux stimulus externes.

Le modèle des cas d'utilisation comprend trois éléments : *i*) les acteurs; *ii*) l'application; *iii*) les cas d'utilisation. Les fonctionnalités de l'application sont exprimées par les cas d'utilisation via les interactions des acteurs avec le système. On représente ces fonctionnalités par des acteurs déclenchant des cas d'utilisation à l'intérieur de l'application.

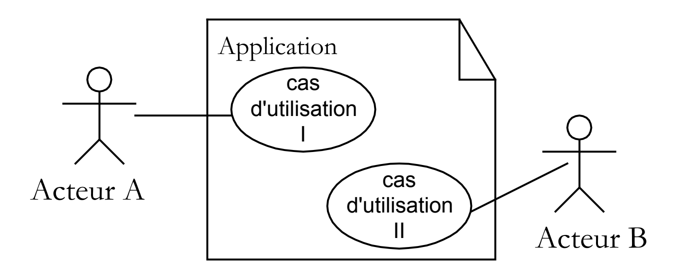


Figure 71 Modèle de cas d'utilisation.

Acteur : Représente une personne ou un système qui interagit avec l'application. Les acteurs sont donc des intervenants réels (utilisateur, technicien opérateur, périphérique, etc.) qui entretiennent des contacts avec l'application considérée. Tout comme dans la vie courante, une même personne peut jouer plusieurs rôles. Ainsi, la même personne peut jouer le rôle de l'utilisateur et de l'opérateur de l'application. Le fait de distinguer les acteurs par leur rôle permet de compartimenter l'analyse des fonctionnalités selon différents points de vue. Enfin, le nom des acteurs dans un diagramme de cas d'utilisation sert à préciser le rôle de l'acteur. Il existe quatre grandes catégories d'acteurs :

- Acteurs principaux : Les personnes qui utilisent les fonctions principales de l'application. Par exemple, les clients d'un guichet bancaire virtuel relié par l'Internet sont des acteurs principaux de l'application « guichet bancaire virtuel ».

- Acteurs secondaires : Les personnes qui administrent ou opèrent l'application. Par exemple, les administrateurs et le personnel de soutien de l'application « guichet bancaire virtuel » sont des acteurs secondaires.

- Matériels externes : Les équipements reliés au système informatique excluant l'ordinateur lui-même. Par exemple, le réseau Internet de l'application « guichet bancaire virtuel » est un matériel externe.

- Autres systèmes : Les autres systèmes informatiques qui entrent en interaction avec l'application. Par exemple, les systèmes de gestion des banques qui réalisent

des transactions avec l'application « guichet bancaire virtuel » sont des systèmes qui agissent comme des acteurs.

5.6.1 DESCRIPTION DES CAS D'UTILISATION

Chaque cas d'utilisation est décrit en langage naturel. Cette documentation sert à établir le pont entre les utilisateurs et les acheteurs de l'application. Dans l'exemple « guichet bancaire virtuel », les banques sont les acheteurs de l'application et les clients de ces banques sont les utilisateurs. Ces deux groupes de personnes n'ont pas nécessairement l'expertise requise pour comprendre les diagrammes UML. La description des cas d'utilisation en langage non technique permet donc de mieux communiquer l'information entre ces deux groupes d'individus. La description des cas d'utilisation comprend six champs :

1. **Nom du cas d'utilisation** : L'identification du cas d'utilisation représentant les cas d'utilisation dans les diagrammes.

2. **Acteurs** : Les acteurs impliqués dans le cas d'utilisation.

3. **Conditions d'entrée** : Les conditions préalables nécessaires pour le déclenchement du cas d'utilisation.

4. **Événements** : La séquence des actions intervenant dans le cas d'utilisation. Les actions sont normalement numérotées.

5. **Conditions de sortie** : Les conditions satisfaites à la fin du cas d'utilisation.

6. **Besoins spéciaux** : Les besoins qui ne sont pas reliés à la fonctionnalité du système.

Voici un cas d'utilisation possible pour l'exemple « guichet bancaire virtuel ».

Nom du cas d'utilisation	`Guichet_Retrait`
Acteurs	Cas d'utilisation invoqué par `Client`. `Systeme_Gestion` valide les requête de `Client`.
Conditions d'entrée	1. `Client` a pointé son fureteur au site Internet approprié. 2. `Client` a donné son nom d'utilisateur. 3. `Client` a donné son NIP (Numéro d'Identification Personnelle).
Événements	4. `Client` choisit le numéro de compte pour effectuer le retrait. 5. `Client` donne le montant de son retrait en multiple de 20$. 6. `Client` appuie sur le bouton OK. 7. `Systeme_Gestion` valide le montant demandé par `Client` avec le solde du compte choisi. `Systeme_Gestion` effectue une réponse en fonction du résultat de la validation.

Conditions de sortie	8. Si la validation par `Systeme_Gestion` donne un résultat positif. La carte de débit virtuelle est augmentée d'un montant équivalent au retrait effectué. 9. Dans le cas contraire, le montant de la carte de débit virtuelle demeura inchangé.
Besoins spéciaux	La communication entre `Client` et `Systeme_Gestion` s'effectue à l'aide d'un canal sécurisé.

5.6.2 SCÉNARIOS

Un cas d'utilisation sert surtout à la compréhension de la situation générale du problème. Ainsi, un cas d'utilisation englobe un ensemble de scénarios qui décrivent des actions concrètes. Les scénarios servent donc à illustrer notre compréhension du problème. Les cas d'utilisation et les scénarios découlant sont normalement validés par les utilisateurs et les acheteurs de l'application.

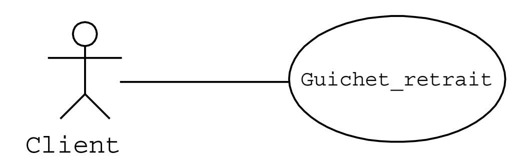
Un scénario comprend seulement trois champs : *i*) Nom du scénario écrit en souligné; *ii*) Noms des instances d'acteur écrits en souligné; *iii*) Séquence détaillée d'événements. Voici un exemple de scénario pour le cas d'utilisation `Guichet_Retrait`.

Nom du scénario	`Guichet_Retrait_BANQUE_BNB`
Acteurs	`Jean : Client` `BNB : Systeme_Gestion`.
Événements	1. `Jean` utilise son NIP pour ouvrir la page « Guichet Virtuel » de la `BNB` (Banque Nationale de Bizouski). 2. `Jean` demande un retrait de 240$ 3. `Jean` confirme le retrait par le bouton OK. 4. `BNB` valide le montant demandé par `Jean` avec le solde du compte choisi. `BNB` effectue une réponse en fonction du résultat de la validation.

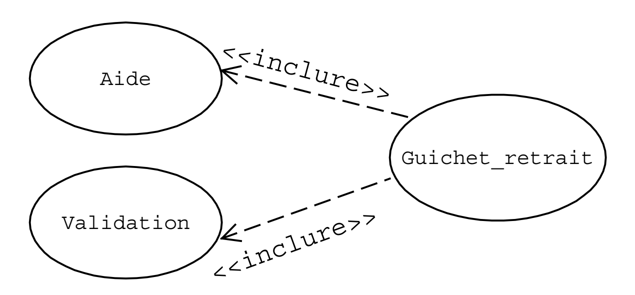
5.6.3 RELATIONS ENTRE CAS D'UTILISATION

Les cas d'utilisation possèdent quatre types de relations. Les acteurs **communiquent** avec les cas d'utilisation. Les cas d'utilisation peuvent **inclure** d'autres cas d'utilisation. Les cas d'utilisation peuvent **étendre** d'autres cas d'utilisation. Enfin, les cas d'utilisation peuvent être une **généralisation** (ou **spécialisation**) de d'autres cas d'utilisation.

- **Communication**. Les acteurs utilisent les cas d'utilisation en échangeant de l'information. Une relation de communication est représentée par une ligne joignant l'acteur et le cas d'utilisation.

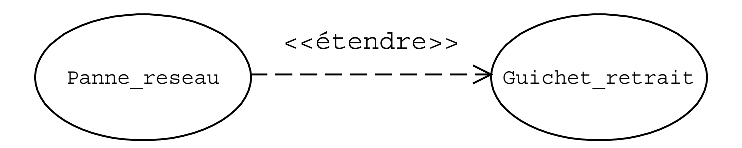


- **Inclusion**. Pour décrire un logiciel complexe, un grand nombre de cas d'utilisation sont nécessaires. Il est possible de simplifier les cas d'utilisation en regroupant des points communs dans quelques cas d'utilisation. Ces cas d'utilisation peuvent alors être inclus dans d'autres cas plus généraux. Par exemple, nous regroupons l'aide en-ligne et validation par le système de gestion de l'application « guichet bancaire virtuel » dans des cas « `Aide` » et « `Validation` ». On peut alors inclure ces deux cas dans « `Guichet_retraite` » réduisant la complexité de ce dernier.



  Dans la description des cas d'utilisation, nous indiquons l'inclusion d'autres cas dans le champ « Besoins spéciaux » s'ils peuvent intervenir n'importe quand durant le déroulement des événements. Si les cas d'utilisation inclus n'interviennent qu'à des moments précis, on peut les indiquer directement dans le déroulement des événements.

- **Extension**. Il s'agit d'un autre moyen de réduire la complexité des cas d'utilisation. Un cas d'utilisation peut être étendu en ajoutant de nouveaux événements. Par exemple, le cas d'utilisation « `Guichet_retrait` » implique la communication entre l'utilisateur et le système de gestion des banques. Le cas d'utilisation « `Panne_reseau` » décrit l'ensemble des événements lorsque le réseau de communication est en panne. Le cas « `Panne_reseau` » étend le cas « `Guichet_retrait` » en séparant le comportement exceptionnel (panne du réseau) du comportement normal (retrait au guichet sans problème).

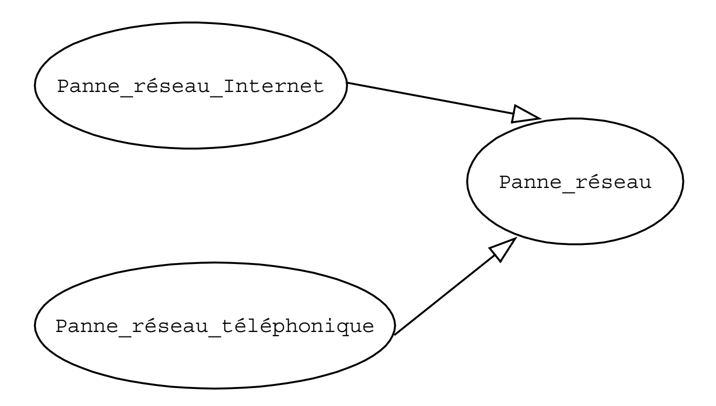


  Le cas « `Guichet_retrait` » est le cas de base qui indique le déroulement normal. Le cas « `Panne_reseau` » indique une exception ou une alternative au déroulement normal.

Nous indiquons dans les conditions d'entrée du cas d'utilisation qui étend (ex : `Panne_reseau`) les cas d'utilisation étendus (ex : `Guichet_retrait`).

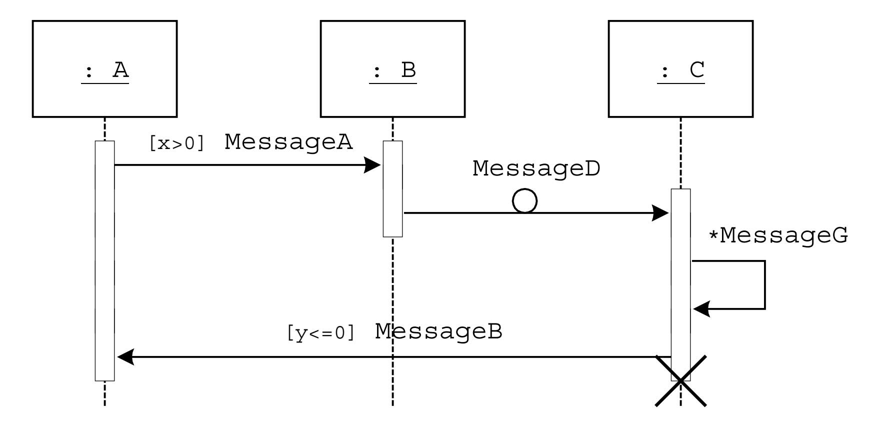
La grande différence entre la relation « `inclure` » et « `étendre` » est l'emplacement de la dépendance. Supposons que nous créons un nouvel ensemble de cas d'utilisation pour l'acteur `Client` de l'application « guichet bancaire virtuel ». Tous ces nouveaux cas d'utilisation doivent inclure le cas « `Aide` ». Par contre, si nous utilisons la relation d'extension, seul le cas « `Aide` » est à modifier pour tenir compte des nouveaux cas d'utilisation de l'acteur `Client`.

- **Généralisation**. Un cas d'utilisation peut ajouter des détails et rendre un autre cas d'utilisation plus spécialisé. Par exemple, le cas « `Panne_réseau` » peut se spécialiser en « `Panne_reseau_Internet` », « `Panne_reseau_téléphonique` ».



5.7 DIAGRAMMES DE SÉQUENCE

Les objets interagissent entre eux par des messages. La réception d'un message par un objet déclenche en lui une opération ou l'envoi de messages vers d'autres objets.



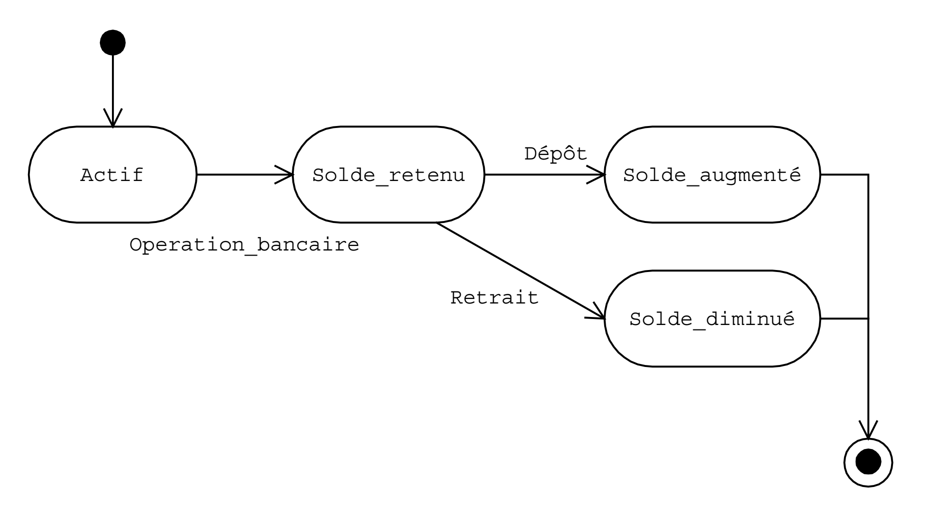
Il est possible d'apposer une condition à l'envoi des messages. Les conditions sont données entre crochets `[ ]`. Par exemple, la condition `[x>0]` MessageA signifie que le message MessageA est envoyé lorsque `x` est positif. Aussi, pour indiquer un

message répétitif, le symbole * est placé devant le nom du message. Les boîtes rectangulaires indiquent la durée active des objets. Enfin le symbole ✕ apposé sur le bas des durées signifie la fin de vie d'un objet.

Les diagrammes de séquence décrivent les interactions entre *n* objets. Normalement, un diagramme de séquence sert à décrire le déroulement des événements d'un cas d'utilisation.

5.8 DIAGRAMMES D'ÉTAT

Les diagrammes d'état servent à montrer la séquence des états traversés par un objet en réponse à des stimulus externes. Un diagramme d'état UML est équivalent à une machine d'état de Mealy (ou Moore). Un état dans ce contexte est une condition satisfaite par un objet. Ainsi, il est une abstraction des valeurs d'attributs d'une classe.

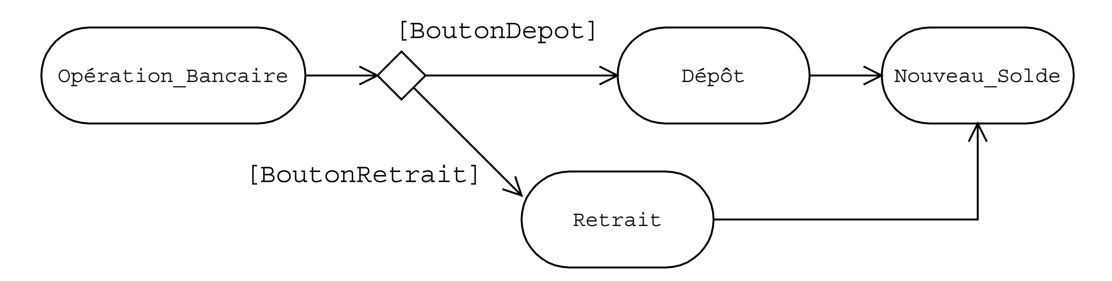


Par exemple, l'objet `Client` est en état `Actif` après avoir donné son NIP dans la page d'accueil de l'application « guichet bancaire virtuel ». L'objet `Client` passe à l'état `Solde_retenu` via la transition `Operation_bancaire` et ainsi de suite. Une transition représente les changements enclenchés par les événements, conditions ou le temps. L'état entrant d'un diagramme d'état est indiqué par un cercle rempli. Les états sortant d'un diagramme d'état sont indiqués par un cercle rempli entouré d'un autre cercle. Les diagrammes d'état sont utilisés pour représenter le comportement d'un objet. Ils servent à montrer explicitement l'ensemble des attributs qui ont un impact sur le comportement d'un seul objet. Ainsi, les diagrammes d'état permettent l'identitifcation des attributs et le raffinement du comportement d'un objet.

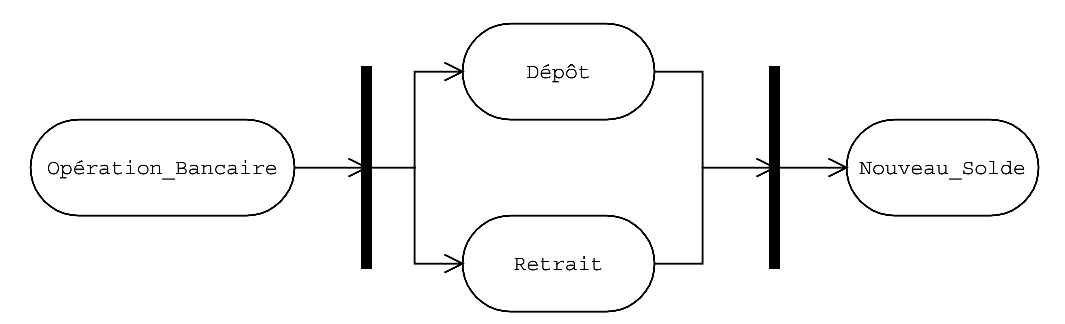
5.9 DIAGRAMMES D'ACTIVITÉS

Les diagrammes d'activités sont des diagrammes d'état dont les états sont des états d'actions. Dans ce type de diagrammes, les transitions sont enclenchées par la fin d'une action associée à l'état. Le nom de l'état dénote donc une action.

Il est possible d'utiliser le branchement dans les diagrammes d'activités. Le branchement représente les transitions alternatives basées sur une condition de l'état d'un objet ou d'un ensemble d'objets. De plus, il est permis de représenter les transitions émanant de plusieurs états ou à multiples destinations.



Dans le cas des branchements, il est nécessaire de donner la raison de branchement entre crochets `[ ]`. Pour les transitions complexes (*n* sources ou *n* destinations), elles dénotent la synchronisation de plusieurs activités (*n* sources) ou elles signifient le partage du travail en plusieurs fils d'exécution (*n* destinations).



5.10 EXEMPLE D'APPLICATION

Nous reprenons l'exemple du système d'ascenseurs de la section 4.8. L'analyse préliminaire effectuée est reprise et exprimée sous forme de diagrammes UML. Les techniques d'analyse utilisées dans l'obtention de ces diagrammes seront discutées dans le chapitre suivant de ce document.

5.10.1 DIAGRAMME ET DESCRIPTION DES CAS D'UTILISATION

D'abord créer le diagramme des cas d'utilisation. Selon l'analyse préliminaire effectuée, le système implique un seul acteur appelé passager. Cet acteur représente l'ensemble des passagers du système d'ascenseurs.

D'un point vue des utilisateurs, l'acteur passager peut : *i*) sélectionner un étage de destination par les boutons B_i situés sur le panneau à l'intérieur des ascenseurs; *ii*) sélectionner un étage de destination par les boutons F_i situés sur les étages du campus; *iii*) entrer ou sortir des ascenseurs à l'arrêt avec les portes ouvertes.

Ces trois cas d'utilisation sont montrés dans la Figure 72. Nous avons ajouté un autre cas d'utilisation libellé « Arrêt urgent » pour représenter la situation anormale de défaillance. Ce dernier vient donc étendre les trois cas d'utilisation de base. Le cas « Arrêt urgent » exprime les événements générés et les conséquences d'une défaillance rencontrée dans le sysème d'ascenseurs.

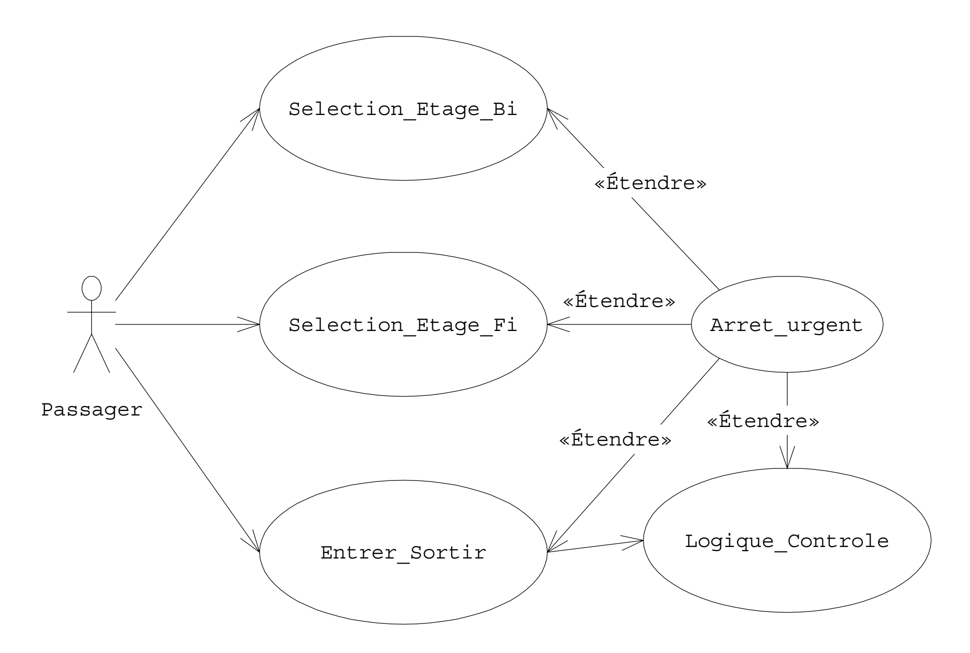


Figure 72 Cas d'utilisation de l'exemple d'application.

Voici la description des cas d'utilisation sous forme tabulaire.

Nom du cas d'utilisation	`Selection_Etage_Bi` (CU_DES1)
Acteurs	Cas d'utilisation invoqué par `Passager`.
Conditions d'entrée	1. En tout moment, lorsqu'il y a défaut de fonctionement, le cas `Arret_urgent` est invoqué. 2. `Passager` est à l'intérieur d'un ascenseur. 3. Le statut de l'ascenseur est différent de REPOS
Événements	4. Le `passager` choisit son étage de destination. 5. Il appuie sur le bouton B_i correspondant de la liste des boutons. Pendant ce temps, l'ascenseur continue son fonctionnement d'une manière ininterrompue.
Conditions de sortie	6. L'étage de destination est pris en compte par le panneau de l'ascenseur et le contrôleur du système. 7. La lumière témoin du bouton sélectionné s'allume. 8. Le compteur de sélections est augmenté de 1.
Besoins spéciaux	NIL

Nom du cas d'utilisation	`Selection_Etage_Fi` (CU_DES2)
Acteurs	Cas d'utilisation invoqué par `Passager`.
Conditions d'entrée	1. En tout moment, lorsqu'il y a défaut de fonctionement, le cas `Arret_urgent` est invoqué. 2. `Passager` situé sur un étage déservi par un ascenseur. 3. Le statut de l'ascenseur est différent de REPOS
Événements	4. Le `passager` décide sur la direction de déplacement (vers le HAUT ou vers le BAS). 5. Il appuie sur le bouton F_i correspondant de la liste des boutons. Pendant ce temps, l'ascenseur continue son fonctionnement d'une manière ininterrompue.
Conditions de sortie	6. L'étage de la requête ainsi que la direction de parcours sont pris en compte par le panneau de l'ascenseur et le contrôleur du système. 7. La lumière témoin du bouton sélectionné s'allume. 8. Le compteur de sélections est augmenté de 1.
Besoins spéciaux	NIL

Nom du cas d'utilisation	`Entrer_Sortir` (CU_DES3)
Acteurs	Cas d'utilisation invoqué par `Passager`.
Conditions d'entrée	1. En tout moment, lorsqu'il y a défaut de fonctionement, le cas `Arret_urgent` est invoqué. 2. `Passager` est à l'intérieur d'un ascenseur ou situé sur un étage déservi par un ascenseur. 3. Le statut de l'ascenseur est à l'ARRÊT. 4. Les portes de l'ascenseur sont ouvertes.
Événements	5. Les utilisateurs de l'ascenseur entrent ou sortent de l'ascenseur. 6. Les boutons Bi et Fi sont désactivés. 7. En même temps, les autres utilisateurs peuvent invoquer les cas d'utilisation « `Selection_Etage_Bi` » et « `Selection_Etage_Fi` ».
Conditions de sortie	8. Les portes de l'ascenseur se ferment. 9. L'ascenseur reçoit la commande de déplacement du contrôleur : Invocation du cas d'utilisation « `Logique_Controle` ».
Besoins spéciaux	NIL

Nom du cas d'utilisation	`Arret_urgent` (CU_DES4)
Acteurs	Cas d'utilisation invoqué automatiquement par le contrôleur du système d'ascenseur.
Conditions d'entrée	1. Une condition de défaut (défaillance) est détectée par le contrôleur.
Événements	2. Désactive les boutons de tous les ascenseurs. 3. Empêcher l'activation des boutons de tous les ascenseurs. 4. Déplacer tous les ascenseurs vers l'étage le plus proche. 5. Déclencher un signal sonore pour indiquer l'ascenseur est en défaillance.

Conditions de sortie	6. Placer les ascenseurs au REPOS (immobiles, portes ouvertes).
Besoins spéciaux	L'arrêt doit demeurer tant et aussi longtemps que la défaillance persiste. Un RAZ du contrôleur permet le rétablissement du système.

Nom du cas d'utilisation	`Logique_Controle` (CU_DES5)
Acteurs	Cas d'utilisation invoqué par le cas « `Entrer_Sortir` ».
Conditions d'entrée	1. En tout moment, lorsqu'il y a défaut de fonctionement, le cas `Arret_urgent` est invoqué. 2. Cas d'utilisation « `Entrer_Sortir` » exécuté avec succès.
Événements	3. L'ascenseur signale au contrôleur qu'il est au REPOS ou à l'ARRÊT. Dans le cas REPOS, la liste de requêtes ne doit pas être vide. 4. L'ascenseur est bloqué en attendant la réponse du contrôleur. 5. Le contrôleur demande à l'ascenseur de lui fournir son étage courant et la liste des requête. 6. L'ascenseur interroge le panneau pour connaître la liste de requêtes et expédie ces informations au contrôleur. L'ascenseur est bloqué en attendant la réponse du contrôleur. 7. Le contrôleur consulte la logique de déplacement et communique la direction et l'étage de destination à l'ascenseur.
Conditions de sortie	8. L'ascenseur change son statut en fonction des données reçues du contrôleur. Il change son statut à REPOS si l'étage de destination est le même que l'étage courant.
Besoins spéciaux	NIL

5.10.2 DIAGRAMME DES CLASSES

Les classes que l'on peut dégager à partir des cas d'utilisation sont illustrées dans le diagramme des classes de la Figure 73. La classe `Bouton` est une classe abstraite. Nous l'utilisons pour dériver deux sous-classes `Bi` et `Fi`. pour représenter les boutons situés dans les ascenseurs et sur les étages. Les boutons `Bi` et `Fi` diffèrent dans leur comportement par le fait que `Bi` représente explicitement les étages tandis que `Fi` représente explicitement les directions de parcours. La classe `ListeBtn` regroupe les boutons `Bi` et `Fi` en une seule collection. La classe `ListeBtn` joue également un autre rôle.

C'est la responsabilité de `ListeBtn` de regrouper les *bons* boutons pour un ascenseur donné. En effet, les ascenseurs du campus ne desservent pas tous les mêmes étages (voir la présentation du problème à la page 172). Ainsi, chaque ascenseur possède une liste particulière de boutons. Enfin, cet ensemble de classes est organisé dans un paquet libellé Bouton.

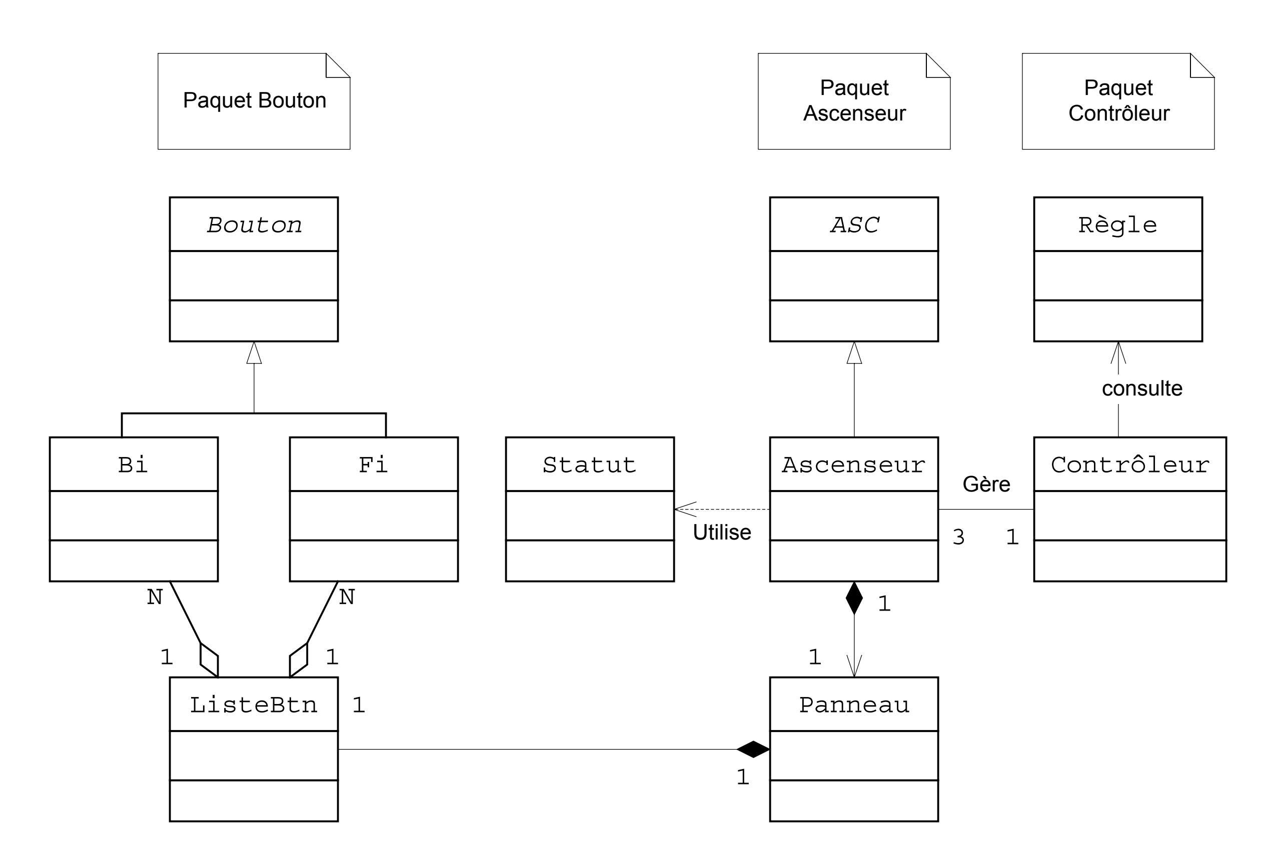


Figure 73 Diagramme des classes de l'exemple d'application.

La classe `Ascenseur` hérite de la classe de base `ASC`. La classe `Ascenseur` est composée d'une classe `Panneau` qui, à son tour, est composée de la classe `ListeBtn` du paquet Bouton. La classe `Ascenseur` dépend également sur la classe `Statut`. Cette dernière représente le statut de fonctionnement de l'ascenseur. Enfin, cet ensemble de classes est organisé dans un paquet libellé Ascenseur.

Le paquet Contrôleur contient deux classes : `Contrôleur` et `Règle`. La classe `Contrôleur` représente le contrôleur du système d'ascenseurs et il gère trois (3) ascenseurs. Les règles qui gouvernent le déplacement des ascenseurs sont représentées par la classe `règle`. Elle forme une dépendance avec la classe `Contrôleur`.

5.10.3 ORGANISATION DES PAQUETS

Nous pouvons organiser les classes de la Figure 73 en paquets. Ces paquets sont : i) Bouton, Ascenseur et Contrôleur. La relation qui existe entre ces paquets est illustrée dans la Figure 74.

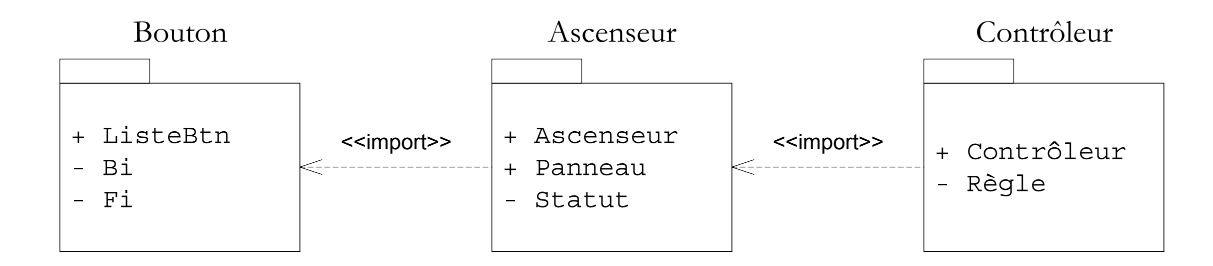
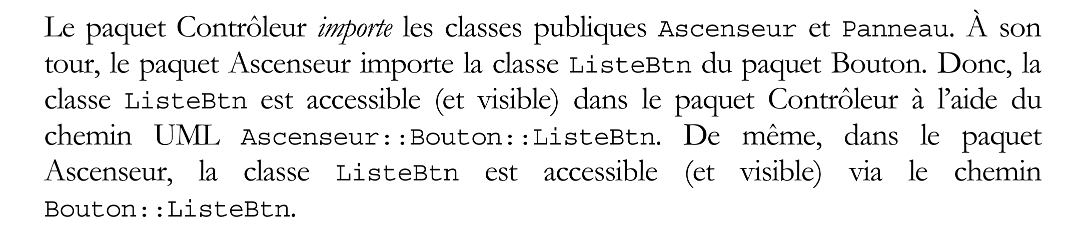


Figure 74 Relations des paquets de l'exemple d'application.

Le symbole + devant le nom d'une classe dans un paquet signifie que la classe est publique et visible par les autres paquets liés par un lien de type « import ». **Attention**! La visibilité des classes dans un paquet n'a pas la même signification que les règles de visibilité du langage C++. ☞


Le paquet Contrôleur *importe* les classes publiques `Ascenseur` et `Panneau`. À son tour, le paquet Ascenseur importe la classe `ListeBtn` du paquet Bouton. Donc, la classe `ListeBtn` est accessible (et visible) dans le paquet Contrôleur à l'aide du chemin UML `Ascenseur::Bouton::ListeBtn`. De même, dans le paquet Ascenseur, la classe `ListeBtn` est accessible (et visible) via le chemin `Bouton::ListeBtn`.

Les relations de la Figure 74 indique également que les classe `Statut`, `Bi` et `Fi` ne sont pas visibles dans le paquet Contrôleur. Autrement dit, les chemins `Ascenseur::Statut`, `Ascenseur::Bouton::Bi` et `Ascenseur::Bouton::Fi` sont des chemins illégaux dans le paquet Contrôleur. Par contre, les relations qui existent entre les paquets UML sont de nature organisationnelle. Elles ne régissent pas la visibilité et l'accessiblité des classes dans une application. Donc, même si les classes `Bi`, `Fi`, `Statut` et `Règle` sont marquées par le symbole `-`, elles sont néanmoins visibles et accessibles dans une application utilisant ces paquets.

5.10.4 DIAGRAMMES DE SÉQUENCE

Les interactions dynamiques des objets de l'exemple sont exprimées à l'aide de diagrammes de séquence. Puisque nous n'avons pas encore déterminé le nom des objets, ils sont identifiés par la notation UML `: nom_de_classe`.

La Figure 75 illustre deux (2) situations. La première partie montre les interactions impliquées dans le cas où un passager choisit un bouton à l'intérieur de l'ascenseur. Le passager sélectionne un étage de destination par le biais des boutons `Bi`. Ce dernier indique à l'objet de type `ListeBtn` qu'il doit effectuer une mise à jour des états du bouton correspondant. Le message de mise à jour est ensuite relayé à l'objet de type `Panneau` de l'ascenseur. Puisque le bouton est de type `Bi`, la mise à jour implique l'activation de la direction HAUT et BAS du bouton correspondant.

Lorsque le passager sélectionne un bouton sur l'étage (un bouton de type `Fi`), il doit décider de la direction associée. Le message de la mise à jour est relayé à l'objet de type `Panneau` comprend la direction du bouton sélectionné.

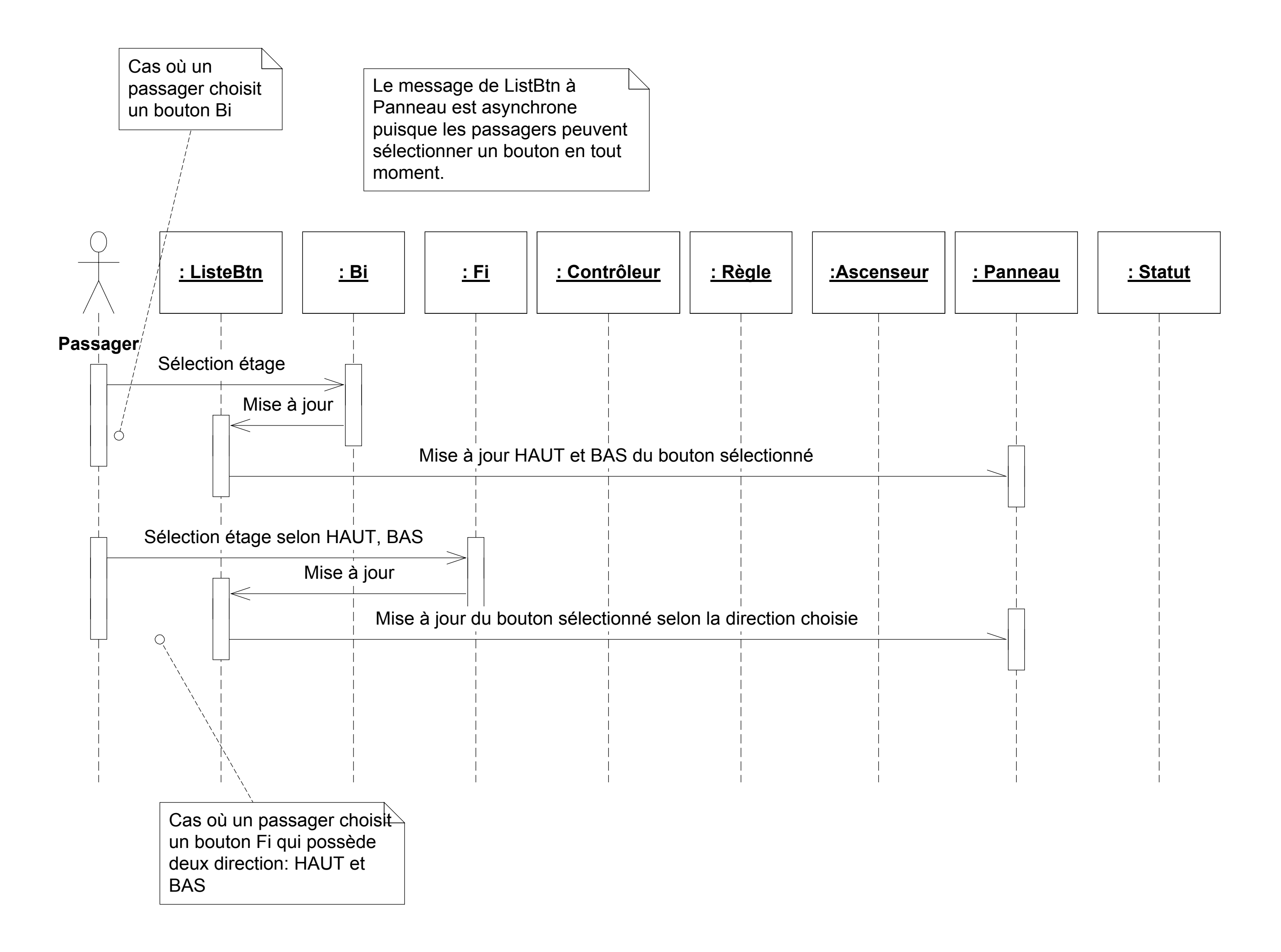


Figure 75 Diagramme de séquence montrant la sélection des boutons.

Noter que le message de l'objet de type `ListeBtn` vers l'objet de type `Panneau` est un message asynchrone. L'utilisation de ce type de message est motivée par le fait que l'activiation des boutons Bi et Fi ne sont pas ordonnée. En plus, le panneau de l'ascenseur peut être sollicité simultannément par le contrôleur des ascenseurs. Les messages asynchones (voir explication à la page 158) conviennent dans de telles situations.

Le diagramme de séquence de la Figure 76 débute par un message synchrone de l'ascenseur vers le contrôleur lui indiquant que l'ascenseur est au repos (ou arrêté). Le message est de type synchrone puisque l'ascenseur ne peut continuer son déplacement tant et aussi longtemps que les consignes du contrôleur ne sont pas obtenues.

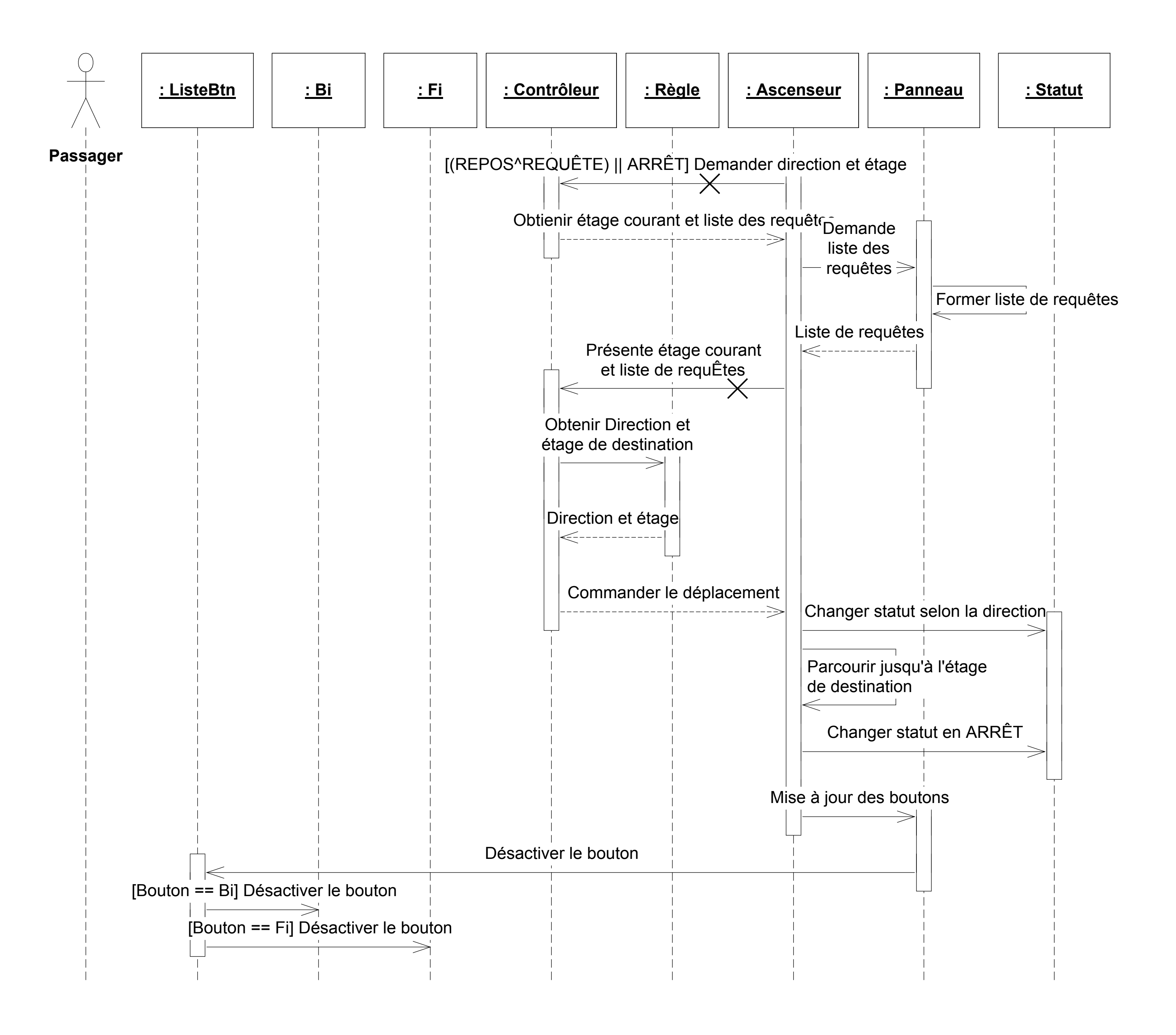


Figure 76 Diagramme de séquence montrant les événements d'un fonctionnement normal des ascenseurs.

Le message en question est une requête pour connaître la direction et l'étage de destination du prochain arrêt de l'ascenseur. Pour répondre à cette requête, le contrôleur demande à l'ascenseur de lui fournir son étage courant et la liste des requêtes en cours. Pour obtenir la liste des requêtes, l'ascenseur passe cette demande à son panneau. Le panneau forme la liste des requêtes et la présente à l'ascenseur qui à son tour retourne les résultats au contrôleur par le biais d'un autre message asynchrone.

Pour décider de l'étage de destination, le contrôleur doit consulter l'objet des règles de déplacement. Ce dernier détermine la prochaine destination de l'ascenseur à l'aide

de l'étage courant et de la liste des requêtes en cours. Après quoi, la commande de déplacement est envoyée à l'ascenseur par le contrôleur.

Avant le parcours vers l'étage de destination, l'ascenseur doit changer son statut. Il en est de même à l'arrivée de la destination. L'ascenseur doit alors indiquer à son panneau d'effectuer la mise à jour de son état. Puisqu'il s'agit de l'arrivée à la destination de l'ascenseur, la mise à jour consiste à désactiver les boutons `Bi` ou `Fi` sélectionnés.

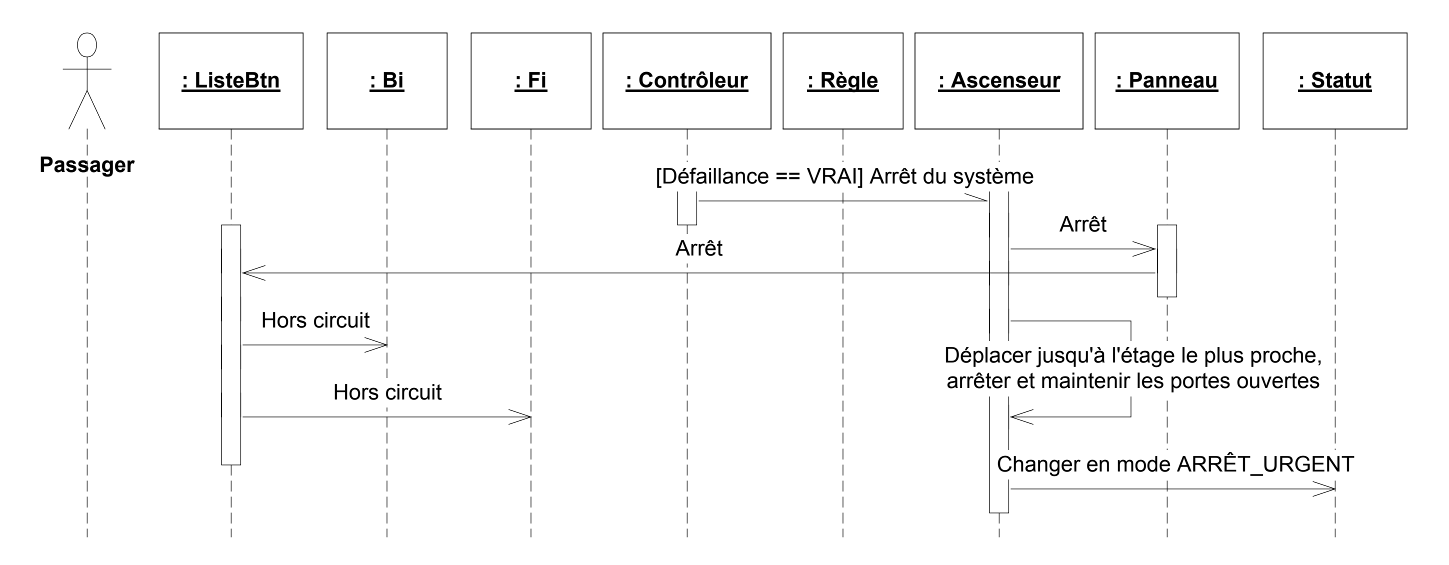


Figure 77 Diagramme de séquence dans le cas d'une détection de défaillance du système d'ascenseurs.

Dans le cas d'un arrêt d'urgence, le contrôleur doit immédiatement aviser les ascenseurs de la situation. Les ascenseurs prennent alors deux actions. La première consiste à désactiver les boutons et empêcher les requêtes de parvenir au panneau. La deuxième action est de transporter les passagers jusqu'à l'étage le plus proche. L'ascenseur doit alors laisser les portes ouvertes indéfiniment. Le statut de l'ascenseur est changé en ARRÊT_URGENT. Noter que ce cas d'arrêt a priorité sur tous les autres cas d'utilisation normale du système des ascenseurs.

5.10.5 DIAGRAMMES D'ÉTAT

Le diagramme d'état de la Figure 78 montre les différents états internes des ascenseurs. Ce diagramme avait été présenté dans le chapitre précédent (voir section 4.8.2). Nous avons ajouté les états supplémentaires pour représenter les changements impliqués lors d'un arrêt d'urgence.

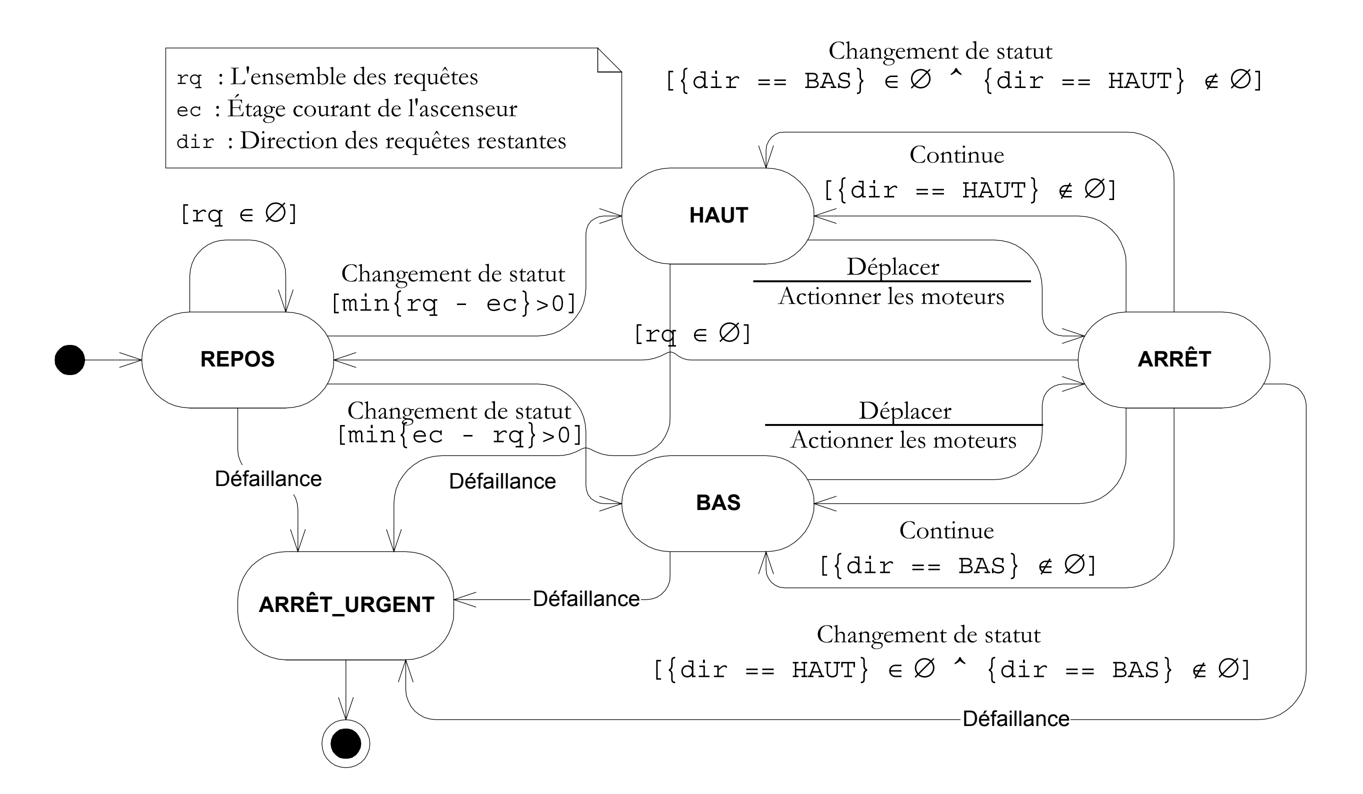


Figure 78 Diagramme d'état des ascenseurs.

Il existe cinq (5) états dans le diagramme de Figure 78. La notation utilisée dans ce diagramme mérite une explication. La condition `[min(rq – ec) > 0]` est lue de la manière suivante : Trouver la plus petite valeur positive de l'opération de soustraction entre l'ensemble des requêtes (exprimées en numéro d'étage) et l'étage courant. Si cette valeur existe cela signifie que la prochaine destination est située à un étage supérieur à l'étage courant. Il en est de même pour la condition `[min{ec – rq} > 0]`. Mais cette fois, c'est la plus petite valeur positive entre l'étage courant et l'ensemble des requêtes. Si cette valeur existe cela signifie que la prochaine destination est située à un étage inférieur à l'étage courant. Il est évident que ces deux conditions sont mutuellement exclusives.

Quant à la condition `[{dir == BAS} ∈ ∅ ∧ {dir == HAUT} ∉ ∅]`, elle sstipule que l'ensemble des requêtes dont la direction est vers le bas est vide ET l'ensemble des requêtes dont la direction est vers le haut n'est pas vide. Cela signifie que l'ensemble des requêtes restantes est dans la direction HAUT. On peut comprendre la condition `[{dir == HAUT} ∈ ∅ ∧ {dir == BAS} ∉ ∅]` de la même façon. Dans le cas de la transition ARRÊT → HAUT `[{dir == HAUT} ∉ ∅]`, l'ascenseur poursuit simplement son chemin dans la même direction afin de desservir les autres destinations qui sont situées à des étages supérieurs à l'étage courant. On peut faire la même remarque pour la transition ARRÊT → BAS `[{dir == BAS} ∉ ∅]`.

Enfin, peu importe l'état où se trouve les ascenseurs, ils transiteront tous vers l'état ARRÊT_URGENT en cas de défaillance du système d'ascenseur.

Les boutons sont à l'état initial ATTENDRE. Après une sélection par des passagers, les boutons transitent dans l'un des trois autres états suivants : SELECTION_HAUT_BAS, SELECTION_HAUT et SELECTION_BAS. L'état SELECTION_HAUT_BAS est réservé pour les boutons de type `Bi` seulement. On retrouve dans le même diagramme d'état un **état composé** représentant les lumières témoins. Les trois états de sélection mentionnés passeront inconditionnellement dans cet état composé. Finalement, en cas de défaillance tous les états passeront dans l'état HORS_CIRCUIT. Par contre, les lumières témoins seront étendues avant de transiter dans cet état de sortie du diagramme.

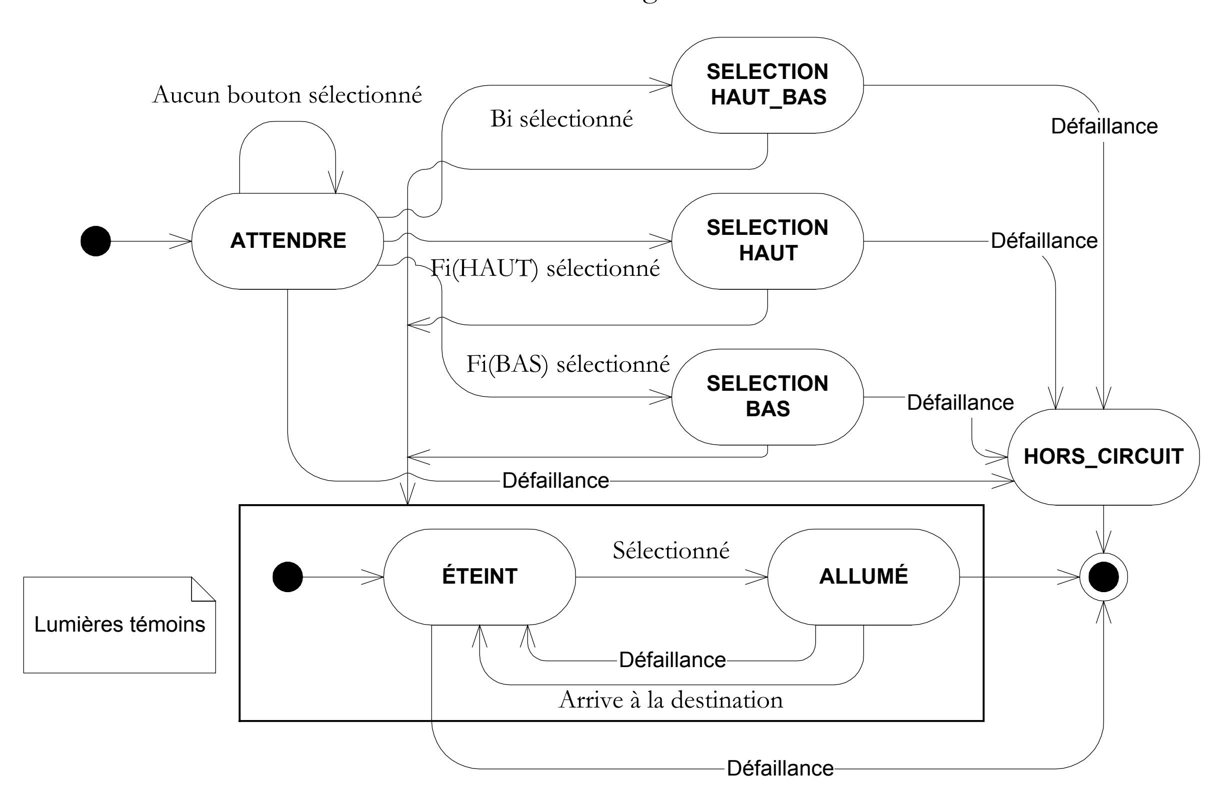


Figure 79 Diagramme d'état des boutons et des lumières témoin.

Le contrôleur possède un diagramme d'état fort simple. Ce diagramme est illustré dans la Figure 80. Le contrôleur est en attente d'une requête. À la réception d'une requête, le contrôleur passe à l'état ACCEPTER puis à l'état TRAITER. La raison de ces deux états est qu'il peut exister plus d'une requête (il y a trois ascenseurs qui ne sont pas nécessairement synchronisés) et on ne peut les traiter simultannément. Donc, on accepte les requêtes mais elles seront traitées après la réception des paramètres nécessaires. Après le traitement d'une requête, le contrôleur passe à l'état COMMANDER qui consiste à envoyer la direction et l'étage de la destination à l'ascenseur. Le contrôleur peut alors basculer à l'état ATTENDRE s'il n'y a plus de requête ou à l'état ACCEPTER s'il y a des requêtes encore en cours. Lors d'une défaillance, le contrôleur passe à l'état ARRÊT et le contrôleur cessera son fonctionnement.

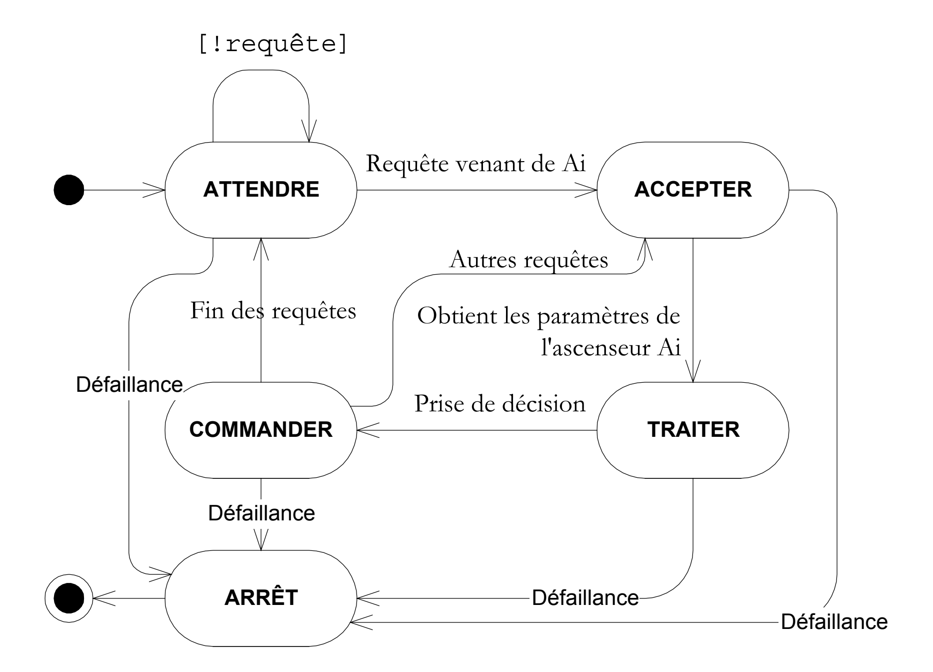


Figure 80 Diagramme d'état du contrôleur.

5.10.6 Autres diagrammes de l'exemple

Pour cet exemple d'application, les autres diagrammes UML seront dégagés lors de l'étude d'analyse et de conception présentée dans les chapitres subséquents de ce document. La raison est que l'explication du contenu de ces diagrammes nécessite des notions appropriées qui ne sont pas encore couvertes dans ce chapitre.

Lecture suggérée

Ce chapitre est une présentation générale de la notation UML.

Les références ci-dessous renferment toutes les informations utiles pour la compréhension du langage UML. Ces références ont été utilisées pour la production de ce chapitre.

[MULL97] Muller, Pierre-Alain, *Instant UML*, Wrox Press, 1997.

Ce livre traite d'une manière pratique les éléments de l'approche orientée objet et surtout la notation UML.

[HTTP01] http://www.omg.org/uml/

La définition, le standard et la pratique de UML sont disponibles chez OMG (*Object Management Group*).

[HTTP02] http://yukon.genie.uottawa.ca/~lavoie/software/uml/

[HTTP03] http://www.celigent.com/omg/umlrtf/tutorials.htm

[HTTP04] http://www.sparxsystems.com.au/UML_Tutorial.htm

[HTTP05] http://uml.free.fr/index-cours.html

Ces références présentent des cours d'UML en-ligne (*on-line tutorial*). Noter que le dernier site (http://uml.free.fr/index-cours.html) est un site en français.

PROBLÈMES

À venir

CHAPITRE 6

UML et C++

« En dedans, une voix s'écrie, "Mon Dieu !" Mais il faut continuer le travailler. On verra le reste plus tard. »

— *Kevin Carter, Photographe de Time Magazine.*

La notation UML exprime d'une manière succincte la modélisation par l'approche orientée objet. Elle donne une vue précise de l'approche tout en conservant une indépendance vis-à-vis les différents langages d'implantation. Or, l'outil premier de la réalisation logicielle demeure les langages de programmation. Inévitablement, la concrétisation d'un projet logiciel passe par la programmation et par extension, par la maîtrise et l'habilité des programmeurs. Il est donc nécessaire de faire un rapprochement entre l'approche orientée objet exprimée sous forme de notation UML et la programmation orientée objet. Dans ce chapitre, nous présentons la correspondance entre le UML et le langage C++.

6. UML ET C++

La correspondance entre la notation UML et le code C++ a été réalisé à l'aide du logiciel gratuit ClassBuilder de Jimmy Venema [HTTP04]. Des résultats semblables peuvent être obtenus par le logiciel commercial Rational Modeler de Rational Rose [HTTP05]. Les commentaires et les points d'ancrage spécifiques au générateur de code de ClassBuilder ont été enlevés afin de faciliter la compréhension. Certaines parties du code ont été modifiées manuellement pour les rendre plus pédagogiques.

6.1 CODE GÉNÉRÉ

Le ClassBuilder, tout comme Rational Rose, permet la génération automatique du code C++ à partir des diagrammes UML. Dans le cas de ClasseBuilder, des macros et des fonctions membres seront ajoutées dans la déclaration des classes pour gérer et maintenir les différentes relations possibles entre les objets.

Les macros de type `RELATION_`*`nom`*`_ACTIVE` et `RELATION_`*`nom`*`_PASSIVE` sont insérées dans la déclaration des classes. Ces maros servent à ajouter des variables et des fonctions membres publiques dans les classes pour gérer et manipuler les objets impliqués dans une relation.

Deux fonctions membres privées sont également insérées dans la définition des classes. Ces fonctions membres ajoutées servent à maintenir la durée de vie de la relation et des objets associés à la relation. La fonction membre privée `ConstructorInclude()` est insérée dans les constructeurs d'une classe. Elle sert à initialiser les variables générées par les macros `RELATION_`*nom*`_ACTIVE` et `RELATION_`*nom*`_PASSIVE`. L'autre fonction membre privée s'appelle `DestructorInclude()` et elle est insérée dans le destructeur des classes. Cette dernière est responsable de l'élimination de la relation et/ou destruction des objets impliqués dans la relation.

Normalement, ces macros et fonctions membres de ClassBuilder font partie de l'implantation des classes et on ne doit pas les modifier.

6.2 CLASSE SIMPLE

ClasseA

Note : Les fonctions membres `ConstructorInclide()` et `DestructorInclude()` sont générées par ClassBuilder pour ses besoins internes.

```
#ifndef _CLASSEA_H
#define _CLASSEA_H
class ClasseA
{
// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:

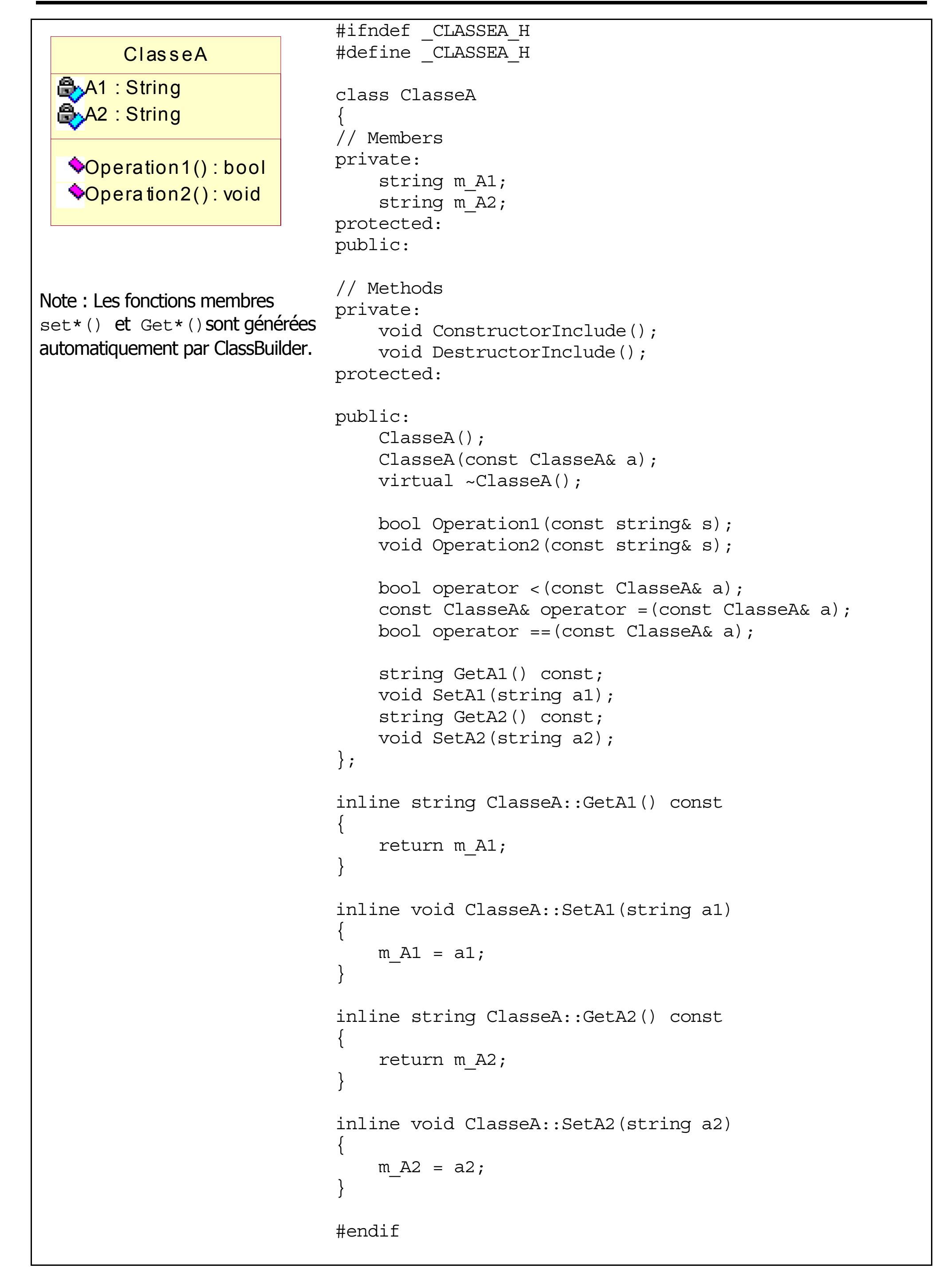
public:
    // constructeur par défaut
    ClasseA();
    // constructeur de copie
    ClasseA(const ClasseA& a);
    // destructeur virtuel
    virtual ~ClasseA();
    // Surcharge des opérateurs
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
#endif
```

Une classe UML qui n'a pas d'attributs et opérations correspond à une classe C++ simple contenant :

- Un constructeur par défaut.
- Un constructeur de copie.
- Un destructeur virtuel (facilite l'héritage éventuel).
- La surcharge des opérateurs =, == et <.

La surcharge de ces opérateurs facilite l'utilisation des classes collection (par exemple STL) pour l'entreposage des objets issus de cette classe.

6.3 Classe avec attributs et opérations



Note : Les fonctions membres `set*()` et `Get*()` sont générées automatiquement par ClassBuilder.

```
#ifndef _CLASSEA_H
#define _CLASSEA_H

class ClasseA
{
// Members
private:
    string m_A1;
    string m_A2;
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();
protected:

public:
    ClasseA();
    ClasseA(const ClasseA& a);
    virtual ~ClasseA();

    bool Operation1(const string& s);
    void Operation2(const string& s);

    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);

    string GetA1() const;
    void SetA1(string a1);
    string GetA2() const;
    void SetA2(string a2);
};

inline string ClasseA::GetA1() const
{
    return m_A1;
}

inline void ClasseA::SetA1(string a1)
{
    m_A1 = a1;
}

inline string ClasseA::GetA2() const
{
    return m_A2;
}

inline void ClasseA::SetA2(string a2)
{
    m_A2 = a2;
}

#endif
```

En plus des fonctions membres d'une classe simple, il existe des fonctions membres Get/Set des attributs `A1` et `A2`. Il est important de prévoir les fonctions membres

d'accès (*accessors*) aux attributs utilisables par les autres classes de l'application. Normalement, les fonctions membres Get/Set sont réalisées sous forme de fonctions inline. Enfin, les deux opérations `Operation1()` et `Operation2()` sont réalisées d'une manière conventionnelle.

6.4 CLASSE PARAMÉTRISÉE

Param

ClasseA

```
#ifndef _CLASSEA_H
#define _CLASSEA_H

template<class Param>
class ClasseA
{
// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA<Param>& a);
    virtual ~ClasseA();

    bool operator <(const ClasseA<Param>& a);
    const ClasseA<Param>& operator =(const
ClasseA<Param>& a);
    bool operator ==(const ClasseA<Param>& a);
};
#endif
```

Une classe paramétrisée est semblable à celle d'une classe simple. Évidemment la différence réside dans le fait qu'une classe paramétrisée est définie par son (ses) paramètre(s). Dans notre cas, le paramètre `Param` est l'élément pivot dans la définition de la classe. On retrouve donc Param dans les fonctions membres impliquant la classe `ClasseA` et dans les constructeurs de celle-ci.

6.5 CLASSE UTILITAIRE

<<utility>>
ClasseA

Operation1(void) : void
Operation2(float) : bool

```
#ifndef _CLASSEA_H
#define _CLASSEA_H

class ClasseA
{
  public:
      static void Operation1 (void);
      static bool Operation2 (float);
};

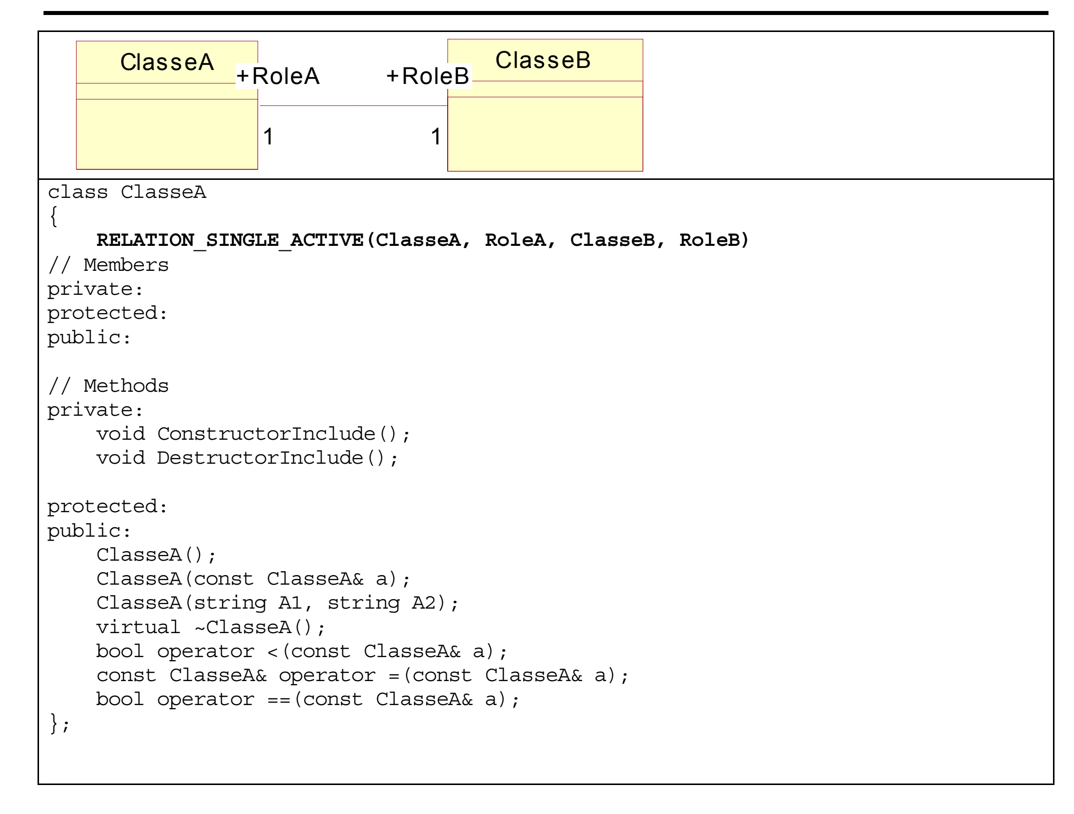
#endif
```

Une classe utilitaire est simplement une classe enveloppe de fonctions. Dans cet exemple, la classe utilitaire `ClasseA` est composée de deux fonctions membres

statiques. Donc, on peut utiliser `Operation1()` et `Operation2()` sans instancier d'objets. Le rôle de la classe est alors un de regroupement logique des fonctions publiques.

6.6 ASSOCIATION 1 À 1

La macro `RELATION_SINGLE_ACTIVE()` est générée et implantée par ClassBuilder. Elle renferme les fonctions membres nécessaires pour manipuler la relation d'association entre les objets de `ClasseA` et les objets de `ClasseB`. Voir explication donnée dans cette sous-section. ☞




```
class ClasseA
{
    RELATION_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)
// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    ClasseA(string A1, string A2);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

La macro `RELATION_SINGLE_PASSIVE()` est générée et implantée par ClassBuilder. Elle renferme les fonctions membres nécessaires pour manipuler la relation d'association entre les objets de `ClasseB` et les objets de `ClasseA`. Voir explication donnée dans cette sous-section. ☞

```
class ClasseB
{
    RELATION_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseB();
    ClasseB(const ClasseB& b);
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

Lorsqu'une relation existe entre deux classes cela signifie que ces classes sont visibles l'une de l'autre. Pour réaliser cette visibilité mutuellement, nous utiliserons un fichier en-tête de la manière suivante.

```
// Fichier en-tête Chapitre6.h
//
// Déclarations anticipée
class ClasseA;
class ClasseB;

// Fichier en-tête de ClassBuilder pour réaliser la relation d'association
// 1 à 1
#include "CB_Single.h"

// Déclaration des classes
#include "ClasseA.h"
#include "ClasseB.h"
```

Le code source du fichier en-tête `Chapitre6.h` utilise les déclarations anticipées pour éviter les problèmes de préséance. En effet dans le fichier `ClasseA.h` la classe `ClasseA` fait référence à la `ClasseB` qui n'est pas encore traitée par le compilateur. Ce problème est réglé par les déclarations anticipées placées au début du fichier `Chapitre6.h`. Ainsi, la déclaration de `ClasseA` est visible par la déclaration de `ClasseB` et vice versa réalisant ainsi la visibilité voulue de la relation d'association.

6.6.1 GESTION D'UNE RELATION D'ASSOCIATION 1 À 1

Le logiciel ClassBuilder applique la technique de classe active – classe passive[1] pour gérer et maintenir une relation entre classes. Dans ClassBuilder, une classe active est celle qui a le contrôle sur la relation. Une classe passive, quant à elle, n'a pas le contrôle direct sur la relation.

La technique de classe active – classe passive est une technique employée par le logiciel ClassBuilder pour gérer une relation entre classes. D'autres logiciels peuvent utiliser des techniques différentes. ☞

La macro `RELATION_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)` et la macro `RELATION_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)` insérées par ClassBuilder joue le rôle de la gestion de la relation d'association entre `ClasseA` et `ClasseB`. Examinons le code généré par ces macros pour mieux comprendre la technique de gestion employée par ClassBuilder.

D'abord, les macros sont contenues dans le fichier d'en-tête `CB_Single.h` de ClassBuilder (répertoire : `X:\ClassBuilder 2.2\Include`). Ce fichier est inclus dans le fichier d'en-tête qui réalise la visibilité des classes (`Chapitre6.h`). Pour voir le code des macros, ouvrir le fichier `CB_Single.h`. Voici la définition de la macro `RELATION_SINGLE_ACTIVE()` :

[1] D'une manière strictement technique, il s'agit d'objet actif et d'objet passif puisque ce sont des objets qui réalisent effectivement les relations.

```
#ifndef CB_PTR
#define CB_PTR(ClassName) ClassName*
#endif

#define RELATION_SINGLE_ACTIVE(ClassFrom, NameFrom, ClassTo, NameTo) \
public:\
    CB_PTR(ClassTo) _ref##NameTo;\
\
public:\
    void Add##NameTo(ClassTo* item);\
    void Remove##NameTo(ClassTo* item);\
    void Replace##NameTo(ClassTo* item, ClassTo* newItem);\
    void Move##NameTo(ClassTo* item);\
    ClassTo* Get##NameTo() const { return _ref##NameTo; };
```

L'expansion de la macro RELATION_SINGLE_ACTIVE() par le compilateur ajoute 1 variable membre et 5 fonctions membres dans la déclaration de la classe active ClasseA. ☞

On constate que cette macro ajoute une (1) variable membre et cinq (5) fonctions membres publiques dans la déclaration de la classe utilisateur (ClasseA dans cet exemple). En utilisant la déclaration

```
RELATION_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)
```

insérée dans la déclaration de ClasseA, voici le rôle de chacun de ces éléments :

- CB_PTR(ClassTo) _ref##NameTo : une variable membre de type pointeur qui pointe vers l'objet passif de la relation. Dans notre exemple, la classe passive est ClasseB. Donc, l'expansion de la macro donne : ClasseB* _ref##RoleB;

- void Add##NameTo(ClassTo* item) : une fonction membre permettant l'assignation d'un objet passif à la variable _ref##NameTo. Cette fonction membre établit donc la relation d'association entre un objet de ClasseA et un objet de ClasseB. Dans notre exemple, la classe passive est ClasseB. Donc, l'expansion de la macro donne : Add##RoleB(ClasseB* item);

- void Remove##NameTo(ClassTo* item) : une fonction member permettant l'élimination d'un objet passif de la variable _ref##NameTo. Cette fonction membre enlève donc la relation d'association entre un objet de ClasseA et un objet de ClasseB. Dans notre exemple, la classe passive est ClasseB. Donc, l'expansion de la macro donne : Remove##RoleB(ClasseB* item);

- void Replace##NameTo(ClassTo* item, ClassTo* newItem) : une fonction membre permettant le replacement d'un objet passif par un autre objet passif. Dans notre exemple, la classe passive est ClasseB. Donc, l'expansion de la macro donne : Replace##RoleB(ClasseB* item, ClasseB* newItem);

- void Move##NameTo(ClassTo* item) : une fonction membre permettant le déplacement d'un objet passif dans un autre objet actif. Cette fonction membre établit donc une nouvelle relation d'association entre un objet actif et un objet passif. De plus, si l'objet passif avait déjà une relation avec un autre objet actif, cette relation est éliminée par cette fonction membre. Dans notre exemple, la classe passive est ClasseB. Donc, l'expansion de la macro donne : Move##RoleB(ClasseB* item);

- `ClassTo* Get##NameTo() const { return _ref##NameTo; }` : une fonction membre permettant d'obtenir un pointeur vers l'objet passif de la relation. Cette fonction membre retourne le pointeur vers l'objet passif. Dans notre exemple, la classe passive est `ClasseB`. Donc, l'expansion de la macro donne : `ClasseB* Get##RoleB() const { return _ref##RoleB; }`

En résumé, la classe active utilise une variable membre et cinq fonctions membres pour gérer une relation d'association avec une classe passive. La variable membre est un pointeur qui pointe vers l'objet passif de la relation. Les fonctions membres réalisent les manipulations de l'objet passif pointé par la variable membre.

L'expansion de la macro `RELATION_SINGLE_PASSIVE()` par le compilateur ajoute 1 variable membre et 1 fonction membre dans la déclaration de la classe passive `ClasseB`. ☞

Voyons maintenant l'expansion de la macro

`RELATION_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)`

insérée dans la déclaration de ClasseB par ClassBuilder.

```
#ifndef CB_PTR
#define CB_PTR(ClassName) ClassName*
#endif

#define RELATION_SINGLE_PASSIVE(ClassFrom, NameFrom, ClassTo, NameTo) \
public:\
    CB_PTR(ClassFrom) _ref##NameFrom;\
\
public:\
    ClassFrom* Get##NameFrom() const { return _ref##NameFrom; };
```

- `CB_PTR(ClassFrom) _ref##NameFrom` : une variable membre de type pointeur qui pointe vers l'objet actif de la relation. Dans notre exemple, la classe active est `ClasseA`. Donc, l'expansion de la macro donne : `ClasseA* _ref##RoleA;`

- `ClassFrom* Get##NameFrom() const { return _ref##NameFrom; }` : une fonction membre permettant d'obtenir un pointeur vers l'objet actif de la relation. Cette fonction membre retourne le pointeur vers l'objet actif. Dans notre exemple, la classe active est `ClasseA`. Donc, l'expansion de la macro donne : `ClasseA* Get##RoleA() const { return _ref##RoleA; }`

En résumé, la classe passive utilise une variable membre et une fonction membre pour établir une relation d'association avec un objet actif. La variable membre est un pointeur qui pointe vers l'objet actif de la relation. La fonction membre permet simplement le retour du contenu de la variable membre. Il est donc évident qu'un objet passif dans ce contexte ne comporte pas autant de responsabilités qu'un objet actif.

Attention! Cela ne signifie pas qu'un objet passif est entièrement contrôlé par l'objet actif associé. En effet, on parle ici de contrôle sur la relation seulement. Si on examine le code présenté précédemment, on voit qu'un objet actif peut utiliser les fonctions membres publiques de l'objet passif et vice versa. Dans le cas de l'objet actif, il peut exécuter les fonctions membres de l'objet passif par le biais la fonction membre `Get##NameTo()` qui retourne un pointeur de l'objet passif. Il en est de même pour l'objet passif, il peut exécuter les fonctions membres de l'objet actif associé par le biais de la fonction membre `Get##NameFrom()`.

6.6.2 MAINTIEN D'UNE RELATION D'ASSOCIATION 1 À 1

Une relation d'association entre deux classes implique une durée de vie déterminée des objets issus de ces classes. En effet, la relation d'association existe si et seulement si les deux objets associés existent.

```
// Dans le fichier ClasseA.cpp (objet actif)
ClasseA::ClasseA()
{
    ConstructorInclude();
    // Put in your own code
}

ClasseA::~ClasseA()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseA::ConstructorInclude()
{ INIT_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB) }

void ClasseA::DestructorInclude()
{ EXIT_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB) }
```

Du côté de l'objet actif, le ClassBuilder utilise deux fonctions membres privées `ConstructorInclude()` et `DestructorInclude()` pour maintenir une relation d'association 1 à 1 avec un objet passif. Ces deux fonctions membres privées sont insérées dans les constructeurs et le destructeur de la classe active. Dans le cas d'une classe active faisant partie d'une association 1 à 1, la fonction membre `ConstructorInclude` contient une macro appelée `INIT_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)`. Cette dernière ne fait que mettre à zéro la variable membre `_ref##NameTo`. Quant à la fonction DestructorInclude exécutée dans le destructeur de la classe active, elle contient une macro appelée `EXIT_SINGLE_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)`. Cette dernière enlève l'objet passif de la variable membre `_ref##NameTo` qui représente la relation d'association. Il n'y a pas de destruction d'objet passif. Donc, à la mort de l'objet actif, la relation d'association sera automatiquement éliminée. Du côté de l'objet passif, le ClassBuilder utilise également les fonctions membres privées `ConstructorInclude()` et `DestructorInclude()` pour maintenir une relation d'association 1 à 1 avec l'objet actif.

```
// Dans le fichier ClasseB.cpp (objet passif)
ClasseB::ClasseB()
{
    ConstructorInclude();
    // Put in your own code
}

ClasseB::~ClasseB()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseB::ConstructorInclude()
{ INIT_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB) }

void ClasseB::DestructorInclude()
{ EXIT_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB) }
```

Cette fois, les macros sont `INIT_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)` et `EXIT_SINGLE_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)`. La macro `INIT_SINGLE_PASSIVE` met à zéro sa variable membre `_ref##NameFrom`. La macro `EXIT_SINGLE_PASSIVE` remet simplement à zéro la variable membre `_ref##NameTo` de l'objet actif. Donc, à la mort de l'objet passif, sa relation d'association avec l'objet actif est automatiquement éliminée.

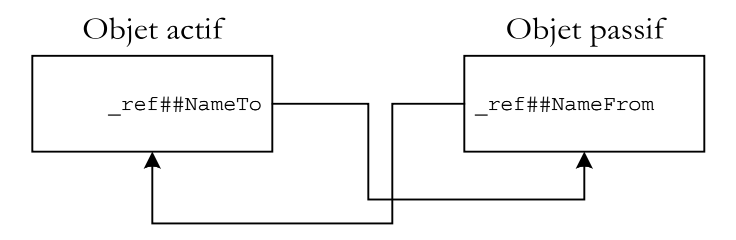
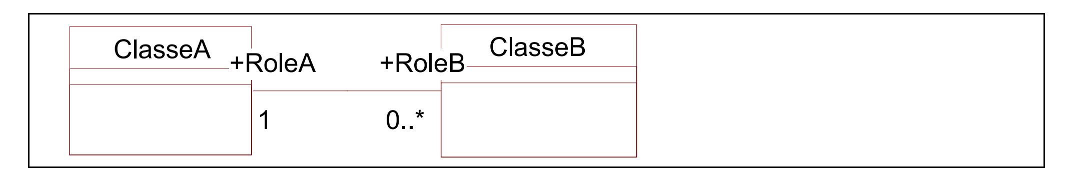


Figure 81 Association 1 à 1 par pointeurs.

Cette technique d'implantation par pointeurs est simple à réaliser et très efficace.

6.7 ASSOCIATION 1 À N



```
class ClasseA
{
    RELATION_NOFILTER_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    ClasseA(string A1, string A2);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

```
class ClasseB
{
    RELATION_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseB();
    ClasseB(const ClasseB& b);
    ClasseB();
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

```
// Fichier d'en-tête Chapitre6.h

// Déclarations anticipées
class ClasseA;
class ClasseB;

// Fichier en-tête de ClassBuilder pour réaliser la relation d'association
// 1 à N

#include "CB_Multi.h"

// Include classes, for declarations
#include "ClasseA.h"
#include "ClasseB.h"

// Déclaration des classes
#include "ClasseA.h"
#include "ClasseB.h"
```

L'implantation d'une association 1 à N ressemble à celle d'une association 1 à 1. Dans le ClassBuilder, la seule différence notable est l'utilisation de la macro `RELATION_NOFILTER_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)` dans la déclaration de `ClasseA` et la macro `RELATION_NOFILTER_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)` dans la déclaration de `ClasseB`. Ces deux macros jouent essentiellement les mêmes rôles que celles de la relation d'association 1 à 1.

6.7.1 GESTION D'UNE RELATION D'ASSOCIATION 1 À N

Dans le contexte d'une association 1 à N, la classe active doit avoir une multiplicité de 1. C'est la classe passive qui porte une multiplicité N. À cause de cela, l'expansion de la macro `RELATION_NOFILTER_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)` produira le code nécessaire pour gérer et maintenir ce type de relation du point de vue de l'objet actif.

```
#ifndef CB_PTR
#define CB_PTR(ClassName) ClassName*
#endif

#define RELATION_NOFILTER_MULTI_ACTIVE(ClassFrom, NameFrom, ClassTo, NameTo) \
public:\
    CB_PTR(ClassTo) _first##NameTo;\
    CB_PTR(ClassTo) _last##NameTo;\
    int _count##NameTo;\
\
public:\
    void Add##NameTo##First(ClassTo* item);\
    void Add##NameTo##Last(ClassTo* item);\
    void Add##NameTo##After(ClassTo* item, ClassTo* pos);\
    void Add##NameTo##Before(ClassTo* item, ClassTo* pos);\
    void Remove##NameTo(ClassTo* item);\
    void RemoveAll##NameTo();\
    void DeleteAll##NameTo();\
    void Replace##NameTo(ClassTo* item, ClassTo* newItem);\
    ClassTo* GetFirst##NameTo() const;\
    ClassTo* GetLast##NameTo() const;\
    ClassTo* GetNext##NameTo(ClassTo* pos) const;\
    ClassTo* GetPrev##NameTo(ClassTo* pos) const;\
    int Get##NameTo##Count() const;\
    void Move##NameTo##First(ClassTo* item);\
    void Move##NameTo##Last(ClassTo* item);\
    void Move##NameTo##After(ClassTo* item, ClassTo* pos);\
    void Move##NameTo##Before(ClassTo* item, ClassTo* pos);\
    void Sort##NameTo(int (*comp)(ClassTo*, ClassTo*));\
    ITERATOR_NOFILTER_MULTI_ACTIVE(ClassFrom, NameFrom, ClassTo, NameTo)
```

Dans le ClassBuilder la multiplicité N des objets passifs est réalisée à l'aide d'une liste chaînée. ☞

Il existe deux variables membres et 19 fonctions membres dans cette macro. Les deux variables membres représentent le premier objet et le dernier objet d'une liste d'objets passifs. Cette liste chaînée permet donc l'implantation de la multiplicité N de la relation d'association. La plupart des fonctions membres de cette macro servent à ajouter, enlever, remplacer, déplacer et parcourir des objets passifs de la liste chaînée.

Une fonction membre particulière est à noter. La fonction membre

```
void Sort##NameTo(int (*comp)(ClassTo*, ClassTo*));
```

permet le triage des objets passifs selon un critère représenté par une fonction passée en paramètre. Cette fonction de comparaison possède la signature suivante :

```
int fnct_comp(ClassTo* obj1, ClassTo* obj2);
```

Elle accepte deux pointeurs d'objet passif et retourne une valeur entière. Donc, la comparaison de ces deux objets doit résulter en une valeur positive si `obj1` > `obj2`, en une valeur négative si `obj1` < `obj2` et en une valeur nulle si `obj1` == `obj2`. La sémantique de cette comparaison est déterminée par le programmeur. Autrement, la signification des termes « plus grand que », « plus petit que », « égal à » pour les objets passifs dépendra du concepteur de ces objets. La fonction membre `Sort##NameTo()` ne fait que trier les objets selon la valeur de retour de la fonction de comparaison.

Enfin, il existe une macro imbriqué dans `RELATION_NOFILTER_MULTI_ACTIVE` qui s'appelle `ITERATOR_NOFILTER_MULTI_ACTIVE(ClassFrom, NameFrom, ClassTo, NameTo)`. Cette dernière produira une classe itérateur directement imbriquée dans la déclaration de la classe active (classes imbriquées). Un itérateur permet le parcours d'une collection (une liste dans ce cas-ci) en utilisant une syntaxe d'une manière orientée objet. Au lieu de parcourir la liste des objets passifs par les fonctions membres `GetNext()`, `GetPrev()`, `GetFirst()` et `GetLast()` on peut parcourir les objets passifs par les opérateur ++ et – de l'itérateur (voir la section 2.4 pour un rappel du rôle des itérateurs).

Quant aux objets passifs, la macro `RELATION_MULTI_PASSIVE` produira trois (3) variables membres et une (1) fonction membre dans la déclaration de la classe passive.

```
#ifndef CB_PTR
#define CB_PTR(ClassName) ClassName*
#endif

#define RELATION_MULTI_PASSIVE(ClassFrom, NameFrom, ClassTo, NameTo) \
public:\
    CB_PTR(ClassFrom) _ref##NameFrom;\
    CB_PTR(ClassTo) _prev##NameFrom;\
    CB_PTR(ClassTo) _next##NameFrom;\
\
public:\
    ClassFrom* Get##NameFrom() const { return _ref##NameFrom; };
```

Les variables `_ref##NameFrom`, `_prev##NameFrom`, `_next##NameFrom` formeront un nœud d'une liste chaînée. L'unique fonction membre générée par cette macro permet d'obtenir le pointeur de l'objet actif de l'association. La Figure 82 montre l'utilisation de ces variables membres dans la formation de la liste chaînée :

- La variable `_ref##NameFrom` pointe vers l'objet actif de la relation.
- La viariable `_prev##NameFrom` pointe vers l'objet passif précédent de la liste.
- La variable `_next##NameFrom` pointe vers l'objet passif suivant de la liste.

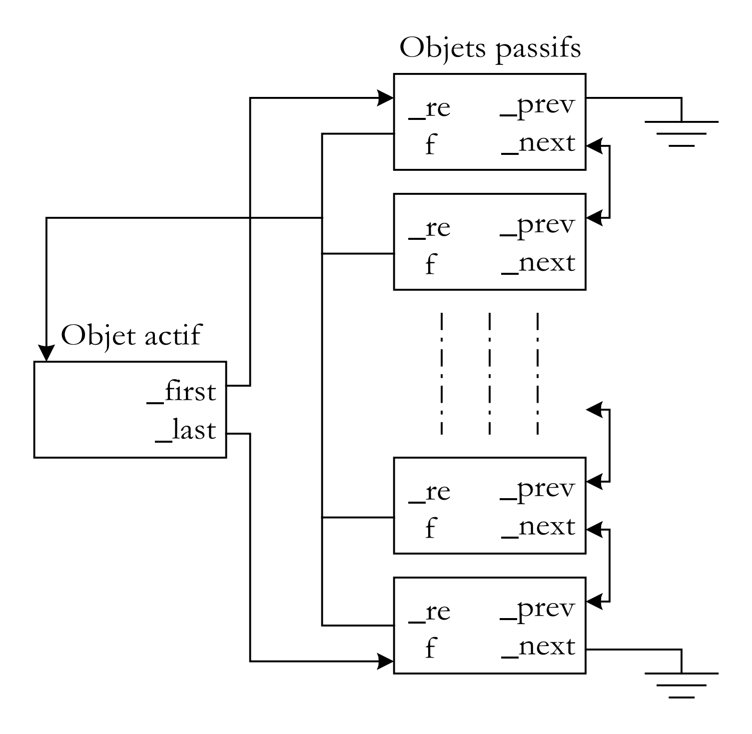


Figure 82 Association 1 à N réalisée par une liste chaînée.

Enfin, la fonction membre `Get##NameFrom()` retourne simplement la valeur de la variable `_ref##NameFrom`.

6.7.2 MAINTIEN D'UNE RELATION D'ASSOCIATION 1 À N

La durée de vie d'une relation d'association 1 à N est la même que celle d'une association 1 à 1. En effet, une association 1 à 1 existe tant et aussi longtemps qu'un objet actif existe et au moins un objet passif existe dans la relation.

```
// Dans le fichier ClasseA.cpp (objet actif)
ClasseA::ClasseA()
{
    ConstructorInclude();
    // Put in your own code
}

ClasseA::~ClasseA()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseA::ConstructorInclude()
{ INIT_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB) }

void ClasseA::DestructorInclude()
{ EXIT_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB) }
```

Attention! Lors de la destruction d'un objet actif, les relations d'association sont enlevées mais les objets passifs ne seront pas détruits.



Du côté de l'objet actif, le ClassBuilder utilise encore une fois les deux fonctions membres privées `ConstructorInclude()` et `DestructorInclude()` pour maintenir une relation d'association 1 à N avec les objets passifs. Ces deux fonctions membres privées sont insérées dans les constructeurs et le destructeur de la classe active. Dans le cas d'une classe active faisant partie d'une association 1 à N, la fonction membre `ConstructorInclude` contient une macro appelée `INIT_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)`. Cette dernière ne fait que mettre à zéro les variables membres de la liste chaînée. Quant à la fonction DestructorInclude exécutée dans le destructeur de la classe active, elle contient une macro appelée `EXIT_MULTI_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)`. Cette dernière parcourt la liste chaînée et enlève les objets passifs de la liste. Il n'y a pas de destruction d'objets passifs. Donc, à la mort de l'objet actif, toutes les relations d'association seront automatiquement éliminées.

Du côté des objets passifs, le ClassBuilder utilise également les fonctions membres privées `ConstructorInclude()` et `DestructorInclude()` pour maintenir une relation d'association 1 à N avec l'objet actif.

```
// Dans le fichier ClasseB.cpp (objets passifs)
ClasseB::ClasseB()
{
    ConstructorInclude();
    // Put in your own code
}

ClasseB::~ClasseB()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseB::ConstructorInclude()
{ INIT_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB) }

void ClasseB::DestructorInclude()
{ EXIT_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB) }
```

Cette fois, les macros sont `INIT_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)` et `EXIT_MULTI_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)`. La macro `INIT_MULTI_PASSIVE`. mettre à zéro les variables membres associées à réalisation de la liste chaînée. La macro `EXIT_MULTI_PASSIVE` enlève l'objet passif (lui-même) de la liste chaînée maintenue par l'objet actif. Donc, à la mort de l'objet passif, sa relation d'association avec l'objet actif est automatiquement éliminée.

6.8 AGRÉGATION 1 À 1

ClasseA +RoleA 1 — 1 +RoleB ClasseB

```
class ClasseA
{
  public:
    //## Constructors (generated)
      ClasseA();
      ClasseA(const ClasseA &right);
    //## Destructor (generated)
      ~ClasseA();

    //## Assignment Operation (generated)
      const ClasseA & operator=(const ClasseA &right);
      bool operator==(const ClasseA &right) const;
      bool operator<(const ClasseA &right) const;

      const ClasseB get_RoleB () const;
      void set_RoleB (ClasseB* value);


  private:  //## implementation
    // Data Members for Associations

      ClasseB* RoleB;
};
```

```
class ClasseB
{
  public:
    //## Constructors (generated)
      ClasseB();
      ClasseB(const ClasseB &right);
    //## Destructor (generated)
      ~ClasseB();

    //## Assignment Operation (generated)
      const ClasseB & operator=(const ClasseB &right);
      bool operator==(const ClasseB &right) const;
      bool operator<(const ClasseB &right) const;

      const ClasseA get_RoleA () const;
      void set_RoleA (ClasseA* value);


  private:  //## implementation
    // Data Members for Associations

      ClasseA* RoleA;
};
```

Le ClassBuilder ne permet pas la réalisation de relations d'agrégation. Cependant, on peut utiliser une relation d'association. En programmation, ces deux relations sont identiques. ☞

Une agrégation représente une relation asymétrique dans laquelle l'une des classes impliquées est plus importante que les autres. Par contre, la durée de vie des classes est indépendante. Dans le ClassBuilder, la relation d'agrégation n'est pas implantée. On peut cependant utiliser les mêmes techniques qu'une relation d'association 1 à 1. L'importance est de réaliser la durée de vie indépendante des objets en agrégation.

Le code ci-dessus est une implantation obtenue par le logiciel Rational Rose. Il s'agit simplement d'une implantation de relation d'association à l'aide de pointeurs.

6.9 AGRÉGATION DIRECTIONNELLE

ClasseA +RoleA 1 — +RoleB 1 ClasseB

```
class ClasseA
{
  public:
    //## Constructors (generated)
      ClasseA();
      ClasseA(const ClasseA &right);
    //## Destructor (generated)
      ~ClasseA();

    //## Assignment Operation (generated)
      const ClasseA & operator=(const ClasseA &right);
      bool operator==(const ClasseA &right) const;
      bool operator<(const ClasseA &right) const;

      const ClasseB get_RoleB () const;
      void set_RoleB (ClasseB* value);


  private:  //## implementation
    // Data Members for Associations

      ClasseB* RoleB;
};
```

```
class ClasseB
{
  public:
    //## Constructors (generated)
      ClasseB();
      ClasseB(const ClasseB &right);
    //## Destructor (generated)
      ~ClasseB();

    //## Assignment Operation (generated)
      const ClasseB & operator=(const ClasseB &right);
      bool operator==(const ClasseB &right) const;
      bool operator<(const ClasseB &right) const;

};
```

La relation d'agrégation directionnelle est implantée de la même façon qu'une relation d'agrégation simple. Cependant, la direction de la flèche nous indique que la classe `ClasseA` n'est pas accessible à partir de la classe `ClasseB`. En fait cette implantation est identique à celle d'une relation d'association directionnelle.

6.10 COMPOSITION

ClasseA +RoleA 1 — +RoleB 1 ClasseB

```
class ClasseA
{
    RELATION_SINGLE_OWNED_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
```

```
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    ClasseA(string A1, string A2);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

```
class ClasseB
{
    RELATION_SINGLE_OWNED_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude(ClasseA* pRoleA);
    void DestructorInclude();

protected:
public:
    ClasseB(const ClasseB& b, ClasseA* pRoleA);
    ClasseB(ClasseA* pRoleA);
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

```
// Fichier d'en-tête (Chapitre6.h)
//
// Déclarations anticipées
class ClasseA;
class ClasseB;

// Fichier en-tête de ClassBuilder pour réaliser la relation de composition
// 1 à 1
#include "CB_SingleOwned.h"

// Déclaration des classes
#include "ClasseA.h"
#include "ClasseB.h"
```

Une relation de composition est encore plus forte qu'une relation d'agrégation. Dans la composition la durée de vie des objets impliqués est normalement identique. Dans le ClassBuilder , les macros `RELATION_SINGLE_OWNED_ACTIVE` et `RELATION_SINGLE_OWNED_PASSIVE` ajouterons les variables membres et les fonctions membres nécessaires pour la réalisation de la relation. Les fonctions membres privées `ConstructorInclude()` et `DestructorInclude()` permettent le maintient de la relation.

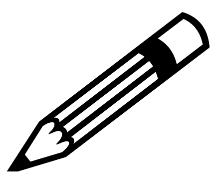
Étudions de plus près le constructeur des classes `ClasseA` et `ClasseB`. Dans la classe active `ClasseA`, le contenu des fonctions `ConstructorInclude()` et `DestructorInclude()` ressemblent à celles des autres relations.

```
// Dans le fichier ClasseA.cpp (objet actif)
ClasseA::ClasseA()
{
    ConstructorInclude();
    // Put in your own code
}


ClasseA::~ClasseA()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseA::ConstructorInclude()
{
    INIT_SINGLE_OWNED_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)
}

void ClasseA::DestructorInclude()
{
    EXIT_SINGLE_OWNED_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)
}
```



Cependant, la macro `EXIT_SINGLE_OWNED_ACTIVE` produit du code qui détruit l'objet passif à la mort de l'objet actif.

```
#define EXIT_SINGLE_OWNED_ACTIVE(ClassFrom, NameFrom, ClassTo, NameTo) \
    if (_ref##NameTo)\
        delete _ref##NameTo;
```

Donc, dans une relation de composition, la destruction de l'objet actif provoque également la destruction de l'objet passif.

```
// Dans le fichier ClasseB.cpp (objet passif)
ClasseB::ClasseB(ClasseA* pRoleA)
{
    ConstructorInclude(pRoleA);
    // Put in your own code
}

ClasseB::~ClasseB()
{
    DestructorInclude();
    // Put in your own code
}

void ClasseB::ConstructorInclude(ClasseA* pRoleA)
{
    INIT_SINGLE_OWNED_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)
}

void ClasseB::DestructorInclude()
{
    EXIT_SINGLE_OWNED_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)
}
```

Il faut ajouter du code manuellement pour forcer la mort simultanée de l'objet actif et de l'objet passif. ☞

Quant à la classe passive, la macro INIT_SINGLE_OWNED_PASSIVE assigne un pointeur de l'objet actif dans une variable membre de l'objet passif dès la construction de l'objet passif. Enfin, dans le ClasseBuilder, la destructeur de l'objet passif contient une macro EXIT_SINGLE_OWNED_PASSIVE qui élimine simplement la relation entre l'objet passif et l'objet actif. Donc, nous devons ajouter du code manuellement pour forcer la coïncidence de la mort de l'objet passif avec la mort de l'objet actif.

6.11 COMPOSITION 1 À N

ClasseA +RoleA 1 ◆———— 0..* +RoleB ClasseB

```
class ClasseA
{
    RELATION_NOFILTER_MULTI_OWNED_ACTIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    ClasseA(string A1, string A2);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

```
class ClasseB
{
    RELATION_NOFILTER_MULTI_OWNED_PASSIVE(ClasseA, RoleA, ClasseB, RoleB)

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude(ClasseA* pRoleA);
    void DestructorInclude();

protected:
public:
    ClasseB(const ClasseB& b, ClasseA* pRoleA);
    ClasseB(ClasseA* pRoleA);
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

```
// Fichier d'en-tête (Chapitre6.h)
//
// Déclarations anticipées
class ClasseA;
class ClasseB;

// Fichier en-tête de ClassBuilder pour réaliser la relation de composition
// 1 à N
#include "CB_MultiOwned.h"

// Déclaration des classes
#include "ClasseA.h"
#include "ClasseB.h"
```

L'implantation de la composition 1 à N est semblable à celle de la composition 1 à 1. La seule différence réside dans le fait qu'une liste chaînée est maintenue dans l'objet actif afin de gérer la multiplicité des objets passifs.

6.12 HÉRITAGE

ClasseA

ClasseB

```
class ClasseA
{
// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

```
class ClasseB
    : public ClasseA
{
// Members
private:
protected:
public:

// Methods
private:
```

```
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseB(const ClasseB& b);
    ClasseB();
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

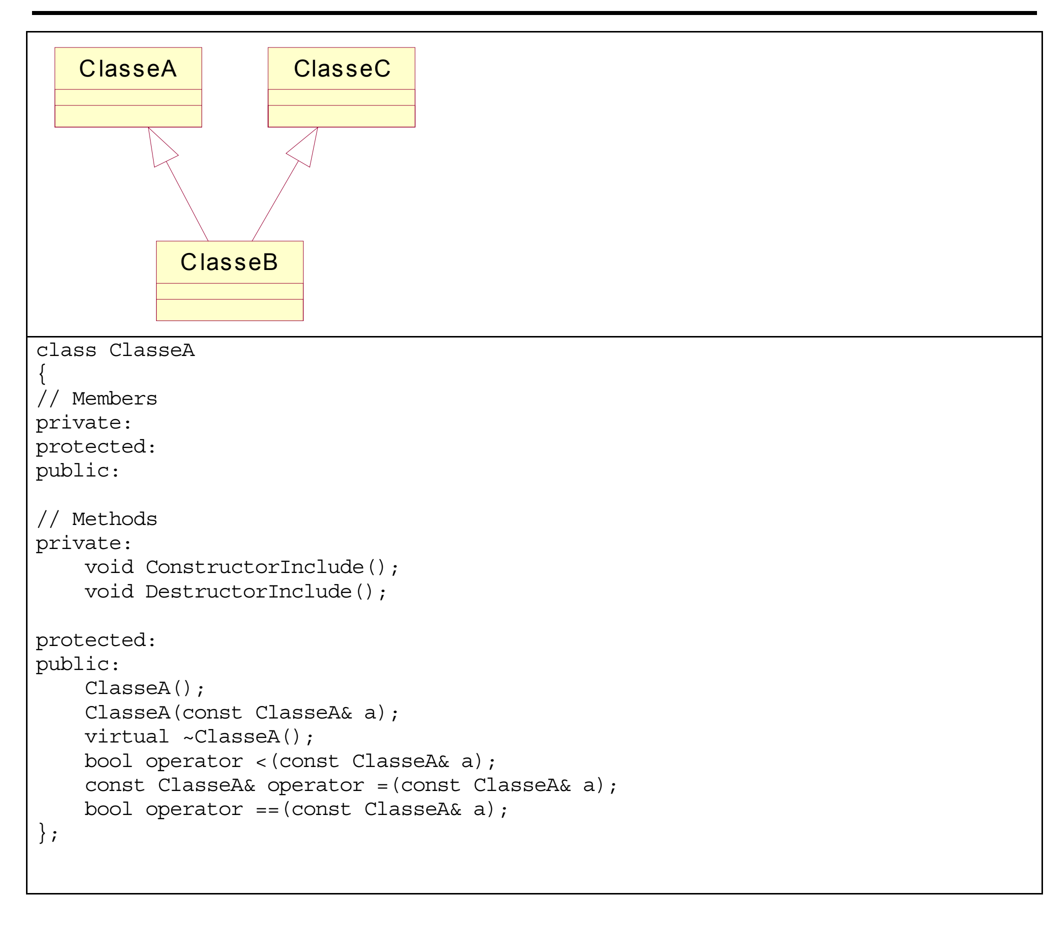
```
// Fichier d'en-tête (Chapitre6.h)

// Déclarations anticipées
class ClasseA;
class ClasseB;

// déclaration des classes
#include "ClasseA.h"
#include "ClasseB.h"
```

Le seul point à remarquer dans la réalisation d'un héritage est l'ajout du mot clé `public` suivi du nom de la classe de base (`public ClasseA`) dans la déclaration de la classe dérivée.

6.13 HÉRITAGE MULTIPLE




```
class ClasseA
{
// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseA();
    ClasseA(const ClasseA& a);
    virtual ~ClasseA();
    bool operator <(const ClasseA& a);
    const ClasseA& operator =(const ClasseA& a);
    bool operator ==(const ClasseA& a);
};
```

```
class ClasseC
{

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseC();
    ClasseC(const ClasseC& c);
    virtual ~ClasseC();
    bool operator <(const ClasseC& c);
    const ClasseC& operator =(const ClasseC& c);
    bool operator ==(const ClasseC& c);
};
```

```
class ClasseB
    : public ClasseA
    , public ClasseC
{

// Members
private:
protected:
public:

// Methods
private:
    void ConstructorInclude();
    void DestructorInclude();

protected:
public:
    ClasseB(const ClasseB& b);
    ClasseB();
    virtual ~ClasseB();
    bool operator <(const ClasseB& b);
    const ClasseB& operator =(const ClasseB& b);
    bool operator ==(const ClasseB& b);
};
```

```
// Fichier d'en-tête (Chapitre6.h)
//
// Déclarations anticipées
class ClasseA;
class ClasseB;
class ClasseC;

// Déclarations des classes
#include "ClasseA.h"
#include "ClasseC.h"
#include "ClasseB.h"
```

Encore une fois, le seul point à remarquer dans la réalisation d'un héritage multiple est l'ajout du mot clé `public` suivi du nom de la classe de base séparée par des virgules (`public ClasseA, ClasseC`) dans la déclaration de la classe dérivée.

6.14 EXEMPLES DE RÉALISATION

La bonne maîtrise technique de la notation UML et du langage de programmation C++ est essentielle à la compréhension de ce chapitre. Des exemples de réalisation sont disponibles sur le site Internet de ce cours à l'adresse (http://www.gpa.etsmtl.ca/cours/gpa789/index.html). Télécharger ces exemples et les étudier attentivement.

LECTURE SUGGÉRÉE

Les références qui ont aidé à la rédaction de ce chapitre sont :

[MULL97] Muller, Pierre-Alain, *Instant UML*, Wrox Press, 1997.

Ce livre traite d'une manière pratique les éléments de l'approche orientée objet et surtout la notation UML.

[HTTP01] http://www.codeproject.com/cpp/oopuml.asp

[HTTP02] http://www.cs.hmc.edu/courses/2000/fall/cs121/turing/umlcc/

[HTTP03] http://www.geocentrix.co.uk/OOP/

[HTTP04] http://home.hetnet.nl/~xvenemaj/ClassBuilder.htm

[HTTP05] http://www.rational.com

PROBLÈMES

À venir

CHAPITRE 7

Stratégies de développement

« Dans les camps et dans les prisons, Ivan Denissovitch s'était désahbitué de prévoir : pour aujourd'hui comme pour dans un an, et comme aussi pour faire vivre les siens. Les chefs s'en occupent à votre place; autant de soucis en moins. »

— *Alexandre Soljenitsyne, Une journée d'Ivan Denissovitch.*

Le processus de développement logiciel est un ensemble d'activités reliées à la création, réa.lisation et maintenance des systèmes logiciels. Il n'existe pas de méthode universelle applicable à tous les processus de développement. Il est donc plus approprié de présenter ce chapitre selon le point de vue de *stratégies* de développement plutôt que de *méthode* de développement. Autrement dit, pour accomplir une activité du processus de développement, un ensemble de conseils et recommandations est présenté. La manière avec laquelle ces recommandations sont appliquées est dépendante du problème à résoudre.

7. STRATÉGIES DE DÉVELOPPEMENT

D'un haut niveau d'abstraction, le processus de développement orienté objet implique les étapes suivantes :

- Planification et élaboration

  Il s'agit de rassembler les ressources humaines et matérielles, définir les exigences, construire les prototypes, etc.

- Construction

  Il s'agit de la réalisation du système logiciel envisagé.

- Déploiement

  L'application et l'implantation du système logiciel dans son environnement d'utilisation.

La Figure 83 présente ces trois grandes étapes du processus de développement.

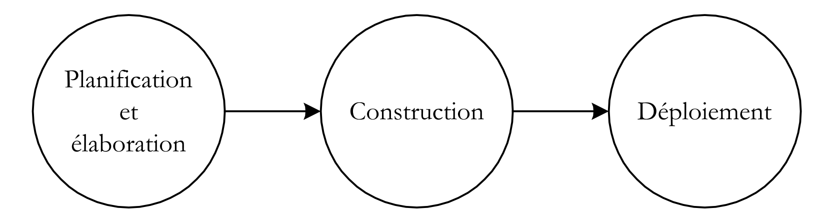
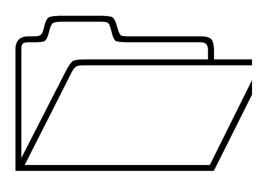


Figure 83 Étapes de développement recommandées.



En pratique, l'approche orientée objet s'inscrit bien dans un processus de développement itératif. Le système logiciel n'est pas construit étape par étape mais bien itération en itération. Un sous-ensemble d'exigences est analysé, des solutions de conceptions sont proposées, la construction et la validation des solutions sont réalisées dans chacune des itérations. De cette façon le système logiciel est construit par incrément et sa complexité devient gérable. De plus, à la fin de chaque itération, il est possible d'apporter des modifications et corriger les erreurs d'analyse et de conception. Cette rétroaction est essentielle puisqu'il est très difficile de concevoir un système jamais commettre des erreurs. Ainsi, les étapes de développement sont modifiées pour tenir compte de la nature itérative des activités nécessaires à la réalisation du système logiciel envisagé.

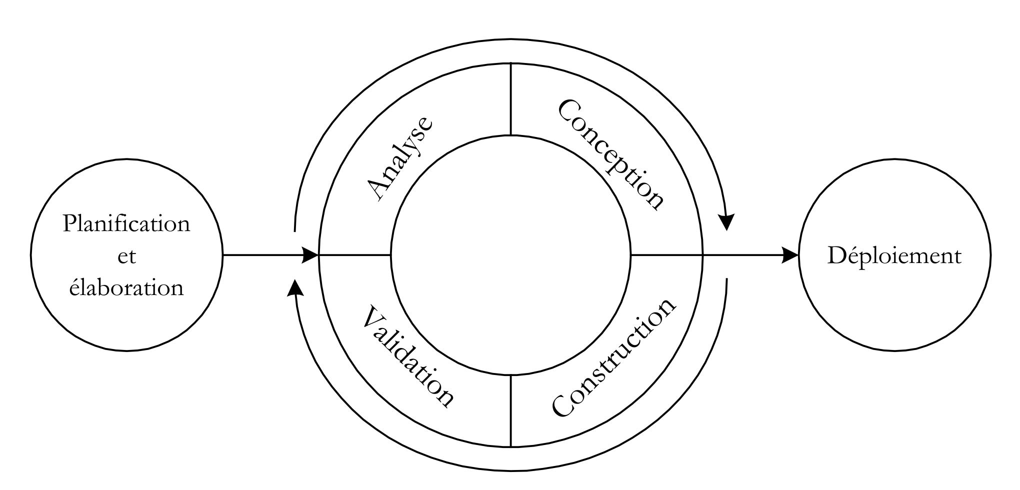


Figure 84 Étapes de développement modifiées.

Dans le modèle de développement de la Figure 84, l'étape de construction est remplacée par un processus itératif comportant quatre phases. Une itération débute par la phase d'analyse et se termine par la phase de validation. Le résultat, à la fin de chaque itération, est un raffinement du système envisagé. Afin de faciliter la gestion du projet de développement, le temps nécessaire pour accomplir une itération doit être borné. Cela signifie qu'un contrôle est exercé pour optimiser l'effort consacré aux activités (des quatre phases). La durée des itérations varie selon le type de projet à réaliser. Elle varie aussi selon l'organisation interne des entreprises (c'est à dire, la culture organisationnelle des entreprises). Il n'existe donc pas de valeurs optimales

pour borner la durée des itérations. Cependant, il est important de limiter la durée des itérations afin de respecter l'échéancier du projet.

7.1 PLANIFICATION ET ÉLABORATION

Cette étape doit précéder toutes les autres étapes du processus de développement. Les activités de cette étape comprennent :

- Création d'un plan de travail

  Le plan de travail doit expliquer le but du projet, les ressources nécessaires (humaines et matérielles), l'horaire des travaux (diagrammes de GANTT), etc.

- Étude préliminaire

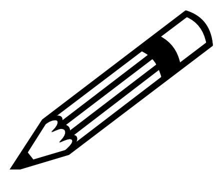
  Il est nécessaire de prendre connaissance de la problématique. Il faut obtenir une bonne compréhension de l'utilité du système logiciel et de son utilisation. Dans la plupart des cas cela implique des rencontres avec les experts et les utilisateurs du domaine d'application.

- Description des exigences

  Décrire en langage naturel ce que doit faire le logiciel. Cette description ne propose pas de solutions. Elle a pour but de présenter, d'une manière informelle, les tâches à accomplir par le système logiciel.

- Dictionnaire des termes

  Le dictionnaire est réalisé sous forme d'un glossaire expliquant le sens des concepts et les mots techniques utilisés dans la documentation de cette étape du processus de développement. Le dictionnaire sert à éliminer, dans la mesure du possible, l'ambiguïté du langage naturel.



En pratique, la planification et l'élaboration des travaux sont des activités réalisées en collaboration avec le client, les experts du domaine d'application et les utilisateurs du système logiciel. La spécification fournie par le client est souvent incomplète. Des rencontres doivent être prévues pour élucider les points obscurs contenus dans la spécification écrite par le client. Les experts du domaine d'application interviennent lorsque des questions techniques sont soulevées. Ces experts sont essentiels au processus de la compréhension de la problématique puisqu'ils détiennent les connaissances techniques et possiblement les solutions de la problématique. Enfin, les utilisateurs sont ceux qui entrent en interaction directe avec le système logiciel. Leurs avis et désirs peuvent rendre le logiciel plus simple d'utilisation mais surtout ils peuvent faciliter l'acceptation du logiciel par les utilisateurs.

Attention! Le prototypage n'est pas toujours requis.

- Le prototypage

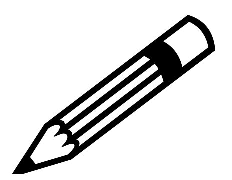
  Dans certaines situations complexes, il est nécessaire de créer des prototypes afin de saisir les nuances et subtilités inhérentes à la problématique. Ces prototypes peuvent être représentés par des maquettes sur papier ou encore par des programmes créés à l'aide de générateurs d'applications.

- Création des cas d'utilisation

  En utilisant la notation UML, produire les cas d'utilisation du système logiciel. Le point de vue adopté est celui d'un généraliste. Autrement dit, ces cas d'utilisation doivent illustrer le principe de fonctionnement du système. Des cas d'utilisation plus élaborés seront produits lors de la phase d'analyse du processus de développement. Finalement, des diagrammes de cas d'utilisation sont établis afin de relier les acteurs du système aux cas d'utilisation.

7.2 DÉVELOPPEMENT ITÉRATIF

Le processus de développement itératif est guidé par les cas d'utilisation créés lors de l'étape de la planification et d'élaboration. Chaque itération doit réaliser les spécifications énoncées dans un sous-ensemble de cas d'utilisation. Parfois les cas d'utilisation sont trop simplistes. Dans de tels cas, nous devons les donner plus de détails réalistes. Parfois les cas d'utilisation sont trop complexes. Dans de tels cas, nous devons les doter de versions simplifiées afin de mieux gérer leur complexité.



Peu importe la nature des cas d'utilisation, il est nécessaire de classifier les cas d'utilisation par une mesure de priorité. Les cas d'utilisation les plus prioritaires doivent être traités dans les premières itérations du processus de développement. **Une stratégie pratique consiste à choisir les cas d'utilisation qui influencent grandement l'architecture du logiciel ou qui comportent des risques (technologiques ou financiers) non négligeables**. La logique est qu'une architecture doit être établie très tôt dans le processus de développement. Les risques, quant à eux, représentent des incertitudes (non pondérables) qu'il faut éliminer avant de pouvoir procéder à des itérations subséquentes.

7.2.1 PHASE D'ANALYSE

Les activités suivantes sont réalisées dans la phase d'analyse du processus de développement.

- Raffinement des cas d'utilisation

  Il s'agit de revoir le (ou les) cas d'utilisation à traiter et apporter des modifications qui s'imposent. Les modifications sont nécessaires s'il manque des détails importants ou si un cas d'utilisation possède une grande complexité.

- Définir le modèle conceptuel

Il s'agit de faire ressortir les concepts véhiculés par les cas d'utilisation raffinés. Les concepts dans ce contexte sont employés comme synonyme de classes. Donc, le modèle conceptuel est l'ensemble cohérent des objets exprimés par les cas d'utilisation en main. Ce modèle conceptuel peut être modifié dans les itérations subséquentes du processus de développement.

- Définir les diagrammes de séquence du système

  Les objets sont en interaction. Nous utilisons les diagrammes de séquence pour illustrer leur comportement dynamique. Ces diagrammes de séquence peuvent être modifiés dans les itérations subséquentes du processus de développement.

- Définir le contrat des opérations

  Les objets entrent en interaction et réalisent ses tâches à l'aide d'opérations. Un objet possède nécessairement des opérations[1][2]. Le contrat est une énumération des opérations de chacun des objets. Les opérations sont identifiées par des phrases en langage naturel. Elles seront formalisées en langage de programmation dans la phase de conception. Le contract des opérations peut être modifié dans les itérations subséquentes du processus de développement.

- Définir les diagrammes d'état

  Les objets sont des entités dynamiques et ils possèdent des états. Ces diagrammes explicitent les états des objets. Il s'agit d'une façon succincte de présenter les opérations des objets et l'effet de ces opérations sur les états des objets. Les diagrammes d'état peuvent être modifiés dans les itérations subséquentes du processus de développement.

- Raffiner le dictionnaire

  Le processus de développement apporte inévitablement de nouveaux concepts et nouvelles entrées dans le dictionnaire des termes. Il est primordial d'effectuer une mise à jour du dictionnaire dans chacune des itérations du processus de développement.

7.2.2 PHASE DE CONCEPTION

Les activités suivantes sont réalisées dans la phase de conception du processus de développement.

- Raffiner les cas d'utilisation

[1] Une opération est identique à une fonction membre en C++ mais à ce stade-ci les opérations sont identifiées par des phrases en langage naturel.

[2] Un objet sans opération n'a pas grande d'utilité.

Dans la phase de conception, le but est de rechercher une solution à la problématique. Les cas d'utilisation doivent être raffinés afin de faire ressortir les concepts qui appartiennent au domaine de la solution.

- Définir la présentation du logiciel

  Le but de cette activité consiste à décrire d'une manière logique l'interface de présentation du logiciel. Dans la plupart des cas, cela signifie la représentation de l'interface graphique. Pour faciliter la compréhension de l'interface graphique, une explication de l'utilisation des éléments de l'interface est nécessaire. La présentation du logiciel peut être modifiée dans les itérations subséquentes du processus de développement.

- Raffiner l'architecture du système

  L'architecture du système logiciel doit évoluer jusqu'à la « maturité » ou jusqu'à une certaine stabilité. Cette évolution est obtenue en la raffinant à chaque itération du processus de développement.

- Définir les diagrammes de classes

  C'est à partir des concepts (classes) dégagés des cas d'utilisation que l'on peut identifier les classes. Le fruit de ce travail fort important est exprimé sous forme de diagrammes de classes. Les diagrammes de classes peuvent être modifiés dans les itérations subséquentes du processus de développement.

Évidemment, cette activité n'est pas obligatoire.

- Définir les bases de données

  La plupart des applications d'envergure utilisent des bases de données. C'est dans la phase de conception que l'on conçoit les tables et les liens de ces bases de données.

7.2.3 PHASE DE CONSTRUCTION

La construction est le codage proprement dit du système logiciel. L'écriture du code source n'est pas une activité isolée. Elle s'appuie plutôt sur le travail réalisé dans les phases d'analyse et de conception. La programmation informatique exige une bonne connaissance de la logique et de la capacité expressive du langage utilisé.

La phase de la construction comprend également le déverminage afin d'éliminer, dans la mesure du possible, les erreurs syntaxiques et logiques du code source. De plus en plus, le déverminage est réalisé par des spécialistes qui ont pour tâches l'examen du code source. Malgré ce fait, le programmeur demeure responsable de l'exactitude du code source et il doit s'assurer de la qualité de sa programmation.

Le code source doit également être documenté. Au minimum, le code source est commenté par le programmeur. Dans la plupart des cas, un cahier de notes est utilisé pour expliquer plus clairement l'implantation des algorithmes et la définition des

structures de données. Enfin, chaque fichier source doit être contrôlé. Ce contrôle est représenté par un numéro de version. Le numéro de version est attribuable manuellement ou automatiquement. L'attribution automatique du numéro de version implique nécessairement un système informatisé de contrôle de version.

7.2.4 PHASE DE VALIDATION

La phase de validation, dans notre contexte, renferme les activités reliées aux tests et aux mesures de la performance du système logiciel. Les tests de logiciel concernent les tests d'unité, les tests d'intégration et du système. Dans les tests d'unité, le logiciel est soumis à des épreuves de validation selon les spécifications du logiciel. Dans les tests d'intégration et de système, le logiciel est soumis à des épreuves de validation selon les spécifications du logiciel **dans son environnement d'utilisation**. Autrement dit, les tests d'unité concernent le logiciel proprement dit. Tandis que les tests d'intégration et de système concernent le logiciel et de son interaction avec l'environnement de fonctionnement. Par exemple, un système de gestion d'inventaire JIT (*Just-In-Time*) est testé avec les entrées réelles de données et est couplé avec la base de données de l'entreprise. Les tests d'intégration et de système sont souvent plus difficiles à réaliser. **La méthode de tests parallèles** est presque toujours utilisée pour faciliter les tests d'intégration et du système. Dans cette méthode, le système logiciel sous test est placé en fonctionnement parallèle avec le système existant. Le système logiciel est certifié **qualité de production** lorsque les tests sont concluants et les résultats sont conformes aux spécifications données.

Dans la littérature classique, les tests du logiciel sont divisés en deux grandes catégories : Les tests dits « boîte noire » et les tests dits « boîte blanche ». La différence essentielle entre ces catégories de tests est la disponibilité du code source. Dans les tests boîte blanche, le code source est accessible aux testeurs. Les procédures de test consistent à identifier les chemins logiques du code source. Autrement dit, le code est testé en divers points et on compare les résultats obtenus avec les résultats attendus. À noter qu'aucun test boîte blanche ne peut être exhaustif. C'est à dire, il n'est pas possible de valider tous les chemins logiques d'un programme. Ceci est causé par l'accroissement exponentiel des chemins logiques qui est une fonction de la taille du programme à tester. Donc, plus la taille du programme est grande plus le nombre de chemins logiques augmente et ce, d'une manière exponentielle.

Les tests boîte noire sont utilisés lorsque le code source n'est pas accessible. Cette situation est présente lorsque le système logiciel est intégré dans son environnement d'utilisation. Souvent les sous-systèmes qui entrent en interaction avec le logiciel sont des sous-systèmes fermés (on ne peut pas obtenir le code source de ces sous-systèmes). Aussi, les tests boîte noire sont utilisés lorsque le logiciel est testé par une firme spécialisée et que l'entente contractuelle ne prévoit pas l'accessibilité du code source[3]. Dans les tests boîte noire, le système logiciel est considéré littéralement

[3] Malheureusement le code source est encore considéré comme le joyau de l'entreprise qu'il faut à tout prix garder le secret. La philosophie OpenSource replace le génie humain en avant plan et non le code source produit.

comme une boîte noire. Les seuls paramètres contrôlables sont : l'entrée des données, les sorties produites par le logiciel et les paramètres de réglage du logiciel. Encore une fois, il n'est pas envisageable de tester toutes les entrées en variant tous les paramètres de réglage. Le même problème d'accroissement exponentiel existe dans les tests boîte noire.

Voici un résumé des activités de cette phase de développement :

- Test d'unité

  Il s'agit d'éliminer, dans la mesure du possible, les erreurs de codage et de fonctionnement du système logiciel.

- Test d'intégration et de système

  Placer le système logiciel dans son environnement d'utilisation et réaliser des tests afin de déceler les erreurs de fonctionnement du système logiciel. Une attention particulière est accordée aux interfaces du système logiciel avec les autres sous-systèmes de son environnement d'utilisation.

- Test de performance

  Il s'agit de valider les limites de performance du système logiciel selon les spécifications du client. L'accent sur mis sur la performance et la fiabilité du système logiciel. Les résultats obtenus serviront à raffiner les algorithmes et les structures de données utilisées dans le système logiciel.

Les méthodes de tests débordent le cadre de ce cours. Pour ceux qui sont intéressés par ce sujet, veuillez consulter les références données à la fin de ce chapitre.

7.3 DÉPLOIEMENT

Cette étape, souvent négligée dans le cadre des études d'analyse et de conception, est une étape cruciale dans le développement d'un produit logiciel. Le déploiement est l'application du système logiciel dans son environnement d'utilisation. Il constitue la mise en place du logiciel chez le client ou l'adoption du système comme outil opérationnel. Par exemple, un système de gestion d'inventaire JIT peut impliquer un grand nombre d'utilisateurs. Le domaine de compétence de ces utilisateurs peut varier selon leur position dans la hiérarchie de l'entreprise. Le système logiciel peut aussi exiger la création de centres de support (nationaux et internationaux). Il est clair que l'étape de déploiement n'est pas simplement le transfert des fichiers exécutables sur l'ordinateur du client !

Voici les activités normalement associées à cette étape :

- Documentation technique

Rassembler tous les diagrammes UML et documents techniques reliés au processus de développement du système logiciel.

- Manuel d'utilisateur

  Présenter l'utilisation du système logiciel en langage naturel. Parfois, il faut présenter le système selon le point de vue du gestionnaire (du système) et selon le point de vue d'utilisateurs qui entrent en interaction avec le système. L'écriture du manuel d'utilisateur est sans doute la partie la plus négligée de cette étape.

- Test d'intégration et de système

  Ces tests sont réalisés dans le milieu de fonctionnement du système. Il est donc normal que l'étape de déploiement soit aussi impliquée dans le test d'intégration et de système.

- Apprentissage et support

  Il s'agit de prévoir l'enseignement de l'utilisation du système par des séances d'apprentissage. Le déploiement d'un système logiciel avec séances d'apprentissage peut augmenter l'acceptation du logiciel par ses utilisateurs. C'est aussi lors de la planification du déploiement que l'on établit la logistique de support. Le personnel au sein de l'entreprise entraîné ou des techniciens de la maison de production du logiciel peut jouer le rôle de personnel de support du système logiciel.

7.4 UTILISATION DES CADRES DE TRAVAIL

De nos jours, le développement des logiciels implique presque toujours l'utilisation d'un cadre de travail (*Framework*). Dans le monde des ordinateurs personnels, les cadres de travail les plus utilisés sont le MFC (*Microsoft Foundation Classes*) de Microsoft, le Visual Component de Borland/Imprise, le PowerPlant de MetroWerks, le Ktk de Linux/KDE. Tous ces cadres de travail ont pour but de simplifier la programmation (système et interface graphique) et de réduire le temps de développement. La plupart de ces cadres de travail enveloppent les appels de système et de l'interface graphique dans des classes structurées.

L'utilisation de ces classes préfabriquées produit parfois des problèmes de « réutilisation ». En effet, une application développée à l'aide de MFC n'est souvent pas compatible avec l'environnement de développement d'un autre vendeur. Lorsqu'il existe un fort couplage entre le cadre de travail et l'application développée, il serait très difficile de réutiliser le code produit.

Le problème de réutilisation est aggravé par la disponibilité des générateurs de code (i.e. AppWizard, ClassWizard de MFC). Ces générateurs de code produisent des sections de code dans lesquelles du code utilisateur peut être ajouté. Ces générateurs de code encouragent l'intégration des routines utilisateurs dans le cadre de travail. Or, le fait de combiner le code utilisateur et le code généré augmente le couplage

entre l'application et le cadre de travail. À l'extrême limite, on ne peut plus distinguer entre le rôle de l'application et le traitement effectué le cadre de travail.

Dans le cas de l'interface graphique, nous pouvons utiliser le concept d'agent intermédiaire pour réduire le couplage entre le cadre de travail et l'application. Par exemple, nous pouvons créer une classe `Affichage` pour représenter l'espace d'affichage de l'application. L'application utilisera cette classe pour réaliser ses sorties vers les fenêtres de l'interface graphique. À son tour, la classe `Affichage` communiquera avec le cadre de travail pour effectuer l'affichage réel.

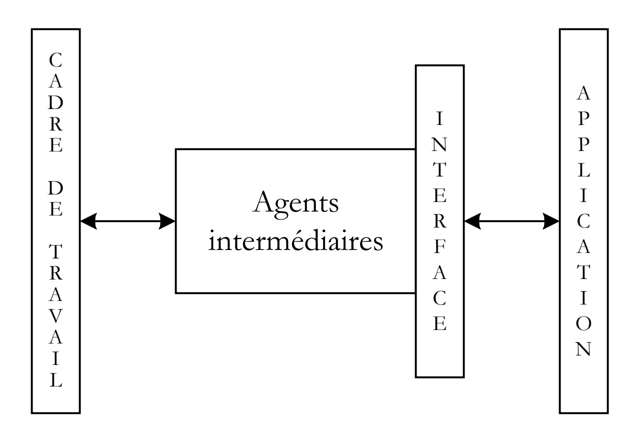


Figure 85 Découplage entre le cadre de travail et l'application afin d'augmenter la réutilisation du code.

De cette façon, l'application n'interagit pas directement avec le cadre de travail. Le code de l'application est donc plus facilement réutilisé dans d'autres projets. Mieux encore, l'application ainsi réalisée peut aisément « portée » à d'autres plate-formes (par exemple, MFC/Windows → PowerPlant/MacOS). En maintenat une interface constante (voir Figure 85), la partie du code à modifier est confinée dans l'implantation des classes qui interagissent avec le cadre de travail.

Les agents intermédiaires sont présents de façon naturelle dans la description des cas d'utilisation. Il suffit de bien noter la description des entrées-sorties, de la gestion des événements et l'utilisation des ressources du système d'exploitation. Par exemple,

- « Produire un rapport sur papier … »

  Un gestionnaire responsable de la gestion de l'imprimante est nécessaire. Il peut utiliser les services du cadre de travail pour écrire les données au périphérique. L'objet `Rapport` utilisera cette classe pour imprimer son contenu.

- « Présenter le graphe d'utilisation … »,

  Un gestionnaire responsable de la gestion de l'affichage est nécessaire. Il peut utiliser les services du cadre de travail pour écrire les données au périphérique. L'objet `Graphe` utilisera cette classe pour afficher son contenu.

- « Démarrer simultanément le chiffrement des messages et leur transfert vers l'ordinateur de destination … »

Ce cas implique la communication inter-ordinateur. La plupart des cadres de travail offrent des classes proxy pour représenter les points de communication (client/serveur). Nous pouvons envelopper ces classes proxy par des classes légères[4] pour diminuer la dépendance de l'application vis-à-vis des cadres de travail. Le transfert des données passera par ces classes légères au lieu d'utiliser directement les classes proxy des cadres de travail.

- « … doit pouvoir décoder plus d'un fichier MP3 à la fois … »

Cette description indique le traitement multi-fils (*multithreading*). Un gestionnaire de fils d'exécution est nécessaire. Il peut utiliser les services du cadre de travail ou directement l'API d'une bibliothèque mutli-fils pour gérer les fils d'exécution de l'application.

- « L'utilisateur peut alors enregistrer les transactions sur disque … »

Nous pouvons considérer cette description de plusieurs façons. Si les transactions sont considérés comme des données simples (i.e. structures avec des champs) alors un simple gestionnaire de sauvegarde est nécessaire. L'application utilisera ce gestionnaire pour enregistrer les données sur disque. Si les transactions sont considérés comme des objets et que le désire réel est l'enregistrement des objets sur disque, alors il serait nécessaire de créer des objets dits **persistants**. La persistance dans ce contexte signifie qu'un objet possède une durée de vie qui dépasse celle de l'application. L'implantation des classes dont ses instances sont persistantes n'est pas une tâche facile. Heureusement, la persistance des objets sont déjà réalisée dans la plupart des cadres de travail. Pour diminuer le couplage, il suffit de créer une hiérarchie de classes semblable à celle de la Figure 86.

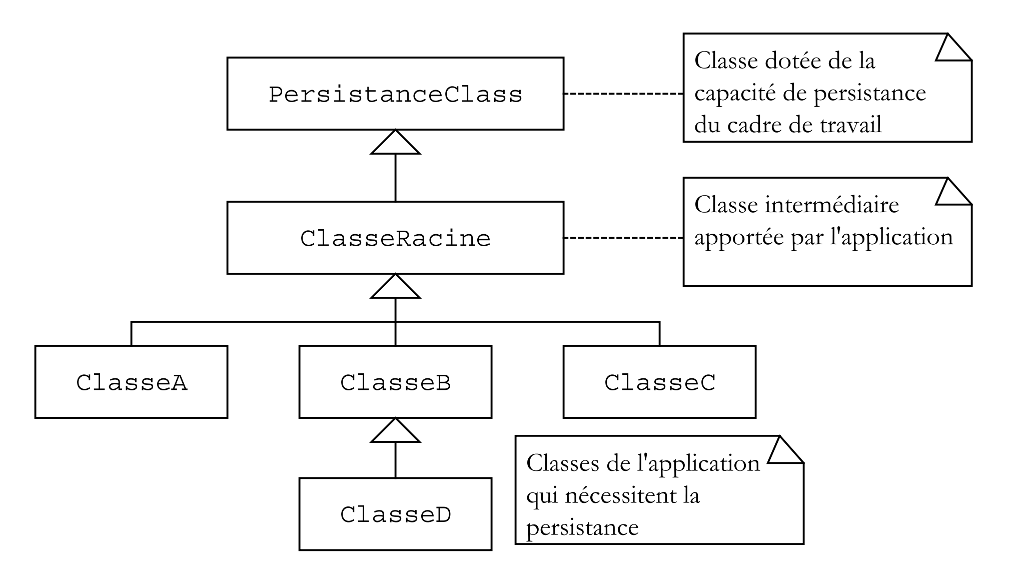


Figure 86 Hiérarchie de classes permettant la persistance et le découplage.

[4] « Légère » dans le sens de « faible complexité ».

Dans la hiérarchie de la Figure 86, la classe `PersistanceClass` est une classe du cadre de travail dotée de la capacité de persistance (la classe `CObjet` de MFC en est un exemple). Les classes `ClasseA` à `ClasseD` sont des classes de l'application nécessitant la capacité de persistance. Ces classes sont dérivées de `ClasseRacine`, également une classe de l'application. Cette dernière, quant à elle, est dérivée de `PersistanceClass`. De cette façon, les classes `ClasseA` à `ClasseD` hériteront les opérateurs et attributs nécessaires pour la réalisation de la persistance des objets. Si pour une raison quelconque, nous devrions changer de cadre de travail. Il suffit de changer la dérivation de la classe `ClasseRacine` puisque les sous-classes demeurent indépendantes du cadre de travail.

LECTURE SUGGÉRÉE

Les références qui ont aidé à la rédaction de ce chapitre sont :

[LARM98] Larman, Craig, *Applying UML and Patterns. An introduction to object-oriented analysis and design*, Prentice-Hall, 1998.

[SOMM95] Sommerville, Ian, *Software Engineering*, Addison-Wesley, 1995.

[PRES00] Pressman, R.S., *Software Engineering : A practitionner's approach*, McGraw-Hill, 2000.

[BOEH88] Boehm, B., *A Spiral Model for Software development and enhancement*, *Computer*, vol. 21, no. 5, pp. 61-72, mai 1988.

[KTK00] http://www.linux.org/apps/AppId_5433.html

[POWE00] http://www.metrowerks.com/desktop/mactools/powerplant/

Le livre de Larman présente une approche pratique de l'analyse et conception orientées objet en utilisant la notation UML et des patrons de conception. La présentation de [LARM98] convient donc parfaitement au contenu de ce chapitre. Enfin, cette référence ne comporte pas de programmation C++ proprement dite. Les références de Sommerville et de Pressman sont dédiées au processus du génie logiciel. Elles traitent les activités englobant la planification et la gestion des projets de développement logiciel. Aussi, certaines méthodes de test sont expliquées dans les ouvrages de Sommerville et Pressman. L'article de Boehm [BOEH88] est le premier papier portant sur le modèle de développement en spiral. Ce modèle spiral, sous forme modifiée, est utilisé dans le processus de développement présenté dans ce chapitre. Enfin, les cadres de travail Ktk (Linux) et PowerPlant (MacOS) sont moins connus mais tout aussi int.éressant.

PROBLÈMES

** | 7.1 Définir succinctement en termes pratiques l'analyse et conception orientées objet.

* | 7.2 Quel est le rôle de la notation UML dans le processus du développement orienté objet ?

*** | 7.3 Présenter un survol du processus de développement nécessaire pour un jeu de dés. Voici sa description :

Un jeu impliquant deux dés. Le joueur lance les dés. Si la valeur totale des dés est sept (7), le joueur est proclamé vainqueur.

** | 7.4 Donner les cas d'utilisation de la question 7.3.

** | 7.5 Donner les diagrammes de séquence de la question 7.3.

** | 7.6 Donner le contrat des opérations de la question 7.3.

** | 7.7 Donner le diagramme d'état de la question 7.3.

** | 7.8 Donner le diagramme des classes de la question 7.3.

** | 7.9 Donner le dictionnaire des termes de la question 7.3.

CHAPITRE 8

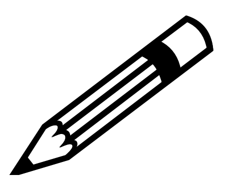
Analyse orientée objet

« … C'est drôle, on la trouve drôle, d'habitude … Elle, à continuait sur le même sujet,, envoye donc, mais j'ai arrêté de l'écouter un moment donné. J'sais pas pourquoi, mais on a toutes arrêté de l'écouter, j'pense… Moé, j'ai fini par me tanner d'attendre sans rien dire. J'me suis levée, tout d'un coup, pis j'ai dit au p'tit : Bon, ben vients-t'en moman est tannée. On r'viendra vendredi. »

— *Michel Tremblay, C't'à ton tour, Laura Cadieux.*

L'analyse orientée objet consiste à dégager un ou des modèles conceptuels représentant le problème à résoudre. Les concepts dans notre cas sont des classes et leur identification est une tâche très importante de l'analyse orientée objet. Ce chapitre présente les stratégies nécessaires pour identifier les concepts, leurs relations, leurs attributs et leurs opérations. Nous préconisons cette identification à partir des cas d'utilisation déjà dégagés.

8. MODÈLES CONCEPTUELS



L'idée première de l'analyse orientée objet est la décomposition d'un problème en concepts (lire classes). Un modèle conceptuel est donc une représentation des concepts exprimés dans le problème à résoudre. Il est nécessaire de bien comprendre qu'un modèle conceptuel est d'abord et avant tout relié aux entités impliquées dans le domaine du problème et non à des entités informatiques.

D'un haut niveau d'abstraction, l'analyse orientée objet explicite les éléments suivants d'un problème :

- Concepts

  Il s'agit l'ensemble des entités exprimé dans les cas d'utilisation. Les cas d'utilisation sont normalement construits à partir des exigences énoncées. **Dans ce chapitre, le mot concept est synonyme du mot classe**.

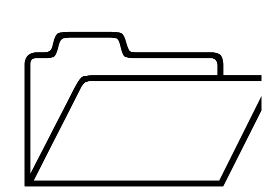
- Associations entre les concepts

Il s'agit de dégager les relations qui existent entre les concepts contenus dans les cas d'utilisation. Ces relations sont importantes puisqu'elles détaillent la structure statique des objets et influencent leur comportement dynamique.

- Attributs des concepts

  Les concepts véhiculés ont des états. Ces états sont exprimés par l'ensemble des valeurs des attributs. Donc, l'identification des attributs est nécessaire pour pouvoir cerner les états des objets.

L'ensemble de ces éléments forme un modèle conceptuel qui est le produit de l'analyse orientée objet. Il est important de noter que le modèle conceptuel n'est pas le modèle de conception. Dans le premier, il s'agit d'une façon de bien saisir la problématique alors que le second est utilisé pour la conception d'un logiciel capable de résoudre le problème posé. Normalement, le modèle conceptuel n'implique que des entités puisées dans le domaine du problème. Dans le modèle de conception il est nécessaire d'introduire des artifices informatiques qui n'ont pas nécessairement de rapport avec le domaine du problème.



Un concept est une idée, une chose ou un objet (dans le sens usuel du terme). Dans l'analyse orientée objet, nous considérons un concept en terme de [MART95, LARM98] :

- Symbole

  Ce sont des mots ou images représentant un concept. Par exemple, un ensemble de gènes regroupés d'un organisme vivant est nommé chronosome. Le mot chromosome est le symbole du concept véhiculé.

- Intension

  Il s'agit de l'ensemble des caractères qui permettent de définir un concept. Par exemple, l'intension du concept chromosome peut être énoncée de la manière suivante : « représente un regroupement de gènes et possède une longueur et un type ».

- Extension

  L'ensemble des exemples applicables à ce concept. Par exemple, l'extension du concept chromosome est l'ensemble de tous les regroupements de tous les gènes d'un organisme vivant.

En pratique, l'extraction des objets passe par l'obtention du symbole et de l'intension des concepts.

8.1 IDENTIFICATION DES CONCEPTS

Cette sous-section présente trois stratégies simples souvent utilisées dans la construction des modèles conceptuels à partir d'un problème bien énoncé. La nomenclature est importante ici. Les stratégies présentées ne sont pas des méthodes. Par conséquent, on ne peut déterminer avec certitude que l'identification effectuée est complète. De plus l'expérience de l'analyste est un facteur non négligeable qui influencera grandement les résultats. Néanmoins, ces stratégies sont simples à comprendre et sont facilement appliquées.

Identification par catégories

Cette stratégie consiste à dresser une liste de choses, d'idées et d'objets (au sens usuel du terme) relevée des cas d'utilisation. Subdiviser le contenu de cette liste en catégories. En voici un exemple :

Catégorie de concepts	Exemples
Objets constituants	Gène, chromosome
Organisation, ensemble	Molécule, protéine
Organisme vivant	Cellule, virus, bactérie
Opération	Mutation, Sélection, accouplement
Processus	Mélange centrifuge, osmose par diffusion
Règle et politique	Distribution uniforme, survie du plus fort

Parfois il est difficile de bien catégoriser les concepts. Certains concepts peuvent appartenir à plus d'une catégorie. Dans de tels cas, il est nécessaire de revoir la liste des catégories en effectuant des fusions ou en scindant une catégorie générale en plusieurs catégories plus spécifiques.

Identification par catégories avec responsabilités

Cette stratégie est semblable à celle présentée plus haut. Pour aider à la tâche nous relevons également les responsabilités des concepts dégagés. Autrement dit, un concept doit avoir des responsabilités envers d'autres concepts ou envers l'environnement de la problématique. Donc, un concept identifié par la catégorisation doit avoir des responsabilités non négligeables qui contribuent à la compréhension de la problématique. Voici un exemple de cette technique d'identification pour la modélisation d'un algorithme génétique dans des problèmes d'optimisation.

Catégorie de concepts	Concept	Responsabilité
Objets constituants	Gène	Représente une variable d'un problème d'optimisation. Il sert d'espace mémoire pour entreposer la variable.
	Chromosome	Regroupe l'ensemble des variables d'un problème d'optimisation.
Organisation	Cellule	Regroupe l'ensemble des chromosomes. Elle constitue l'espace des solutions potentielles de l'algorithme génétique.

Opération	Mutation Sélection Accouplement	Modifie spontanément la valeur d'une ou des variables du problème d'optimisation. Sélectionne des chromosomes pour la reproduction par l'accouplement. Fabrique, à partir des chromosomes sélectionnés, de nouveaux chromosomes. Autrement dit, générer de nouvelles solutions à partir des anciennes. On espère que les nouvelles solutions potentielles seront « meilleures » que les anciennes.
Règle et politique	Distribution uniforme Survie du plus fort	Contribue à la nature stochastique de la mutation et de la sélection. Guide la sélection des chromosomes pour la reproduction. Cette règle stipule que l'on doit sélectionner les « meilleurs » chromosomes pour la reproduction puisque ces chromosomes ont plus de chance de produire des enfants encore plus convenables pour le problème d'optimisation.

Encore une fois, les concepts qui n'ont pas de responsabilité ou qui ont des responsabilités marginales ne seront pas retenus. Ils pourront être transformés en attributs comme nous verrons plus tard dans ce chapitre.

Identification en termes de noms et de phrases

Cette stratégie consiste à relever systématiquement les noms et des phrases importantes contenues dans des cas d'utilisation. Considérer ces noms (et phrases) comme des concepts ou attributs de concepts potentiels. Cependant, nous devons être vigilants puisque le langage naturel (le français par exemple) est truffé de doubles entendre et d'ambiguïté. On ne doit pas utiliser cette stratégie d'une manière mécanique.

L'application de cette stratégie (et de celles présentées précédemment) nécessite la bonne compréhension du problème à résoudre. En effet, les concepts dégagés seront superficiels si nous ne maîtrisons pas le domaine du problème. Il est donc nécessaire de consulter les spécialistes du domaine d'application en cas de difficulté.

D'une manière générale, nous utiliserons les noms et les termes puisés directement dans les cas d'utilisation. Autrement dit, nous ne devons pas inventer des noms et des termes pour satisfaire à nos besoins. Il est également conseillé d'exclure les caractéristiques qui ne sont pas utiles à l'identification des concepts. Autrement dit, certains concepts présents dans les cas d'utilisation peuvent ne pas être utilisés dans le modèle conceptuel. Enfin, il ne faut jamais ajouter un concept dans le modèle conceptuel qui n'a pas d'existence dans les cas d'utilisation. Si le concept est absolument nécessaire alors il faut l'ajouter dans les cas d'utilisation et le valider. Après une validation positive, le concept peut alors être inséré dans le modèle

conceptuel. Le but est d'éviter la création d'inconsistance entre le modèle conceptuel et les exigences énoncées.

8.2 CONSTRUCTION D'UN MODÈLE CONCEPTUEL

Les étapes ci-dessous sont à observer lors de la construction d'un modèle conceptuel.

1. Utiliser l'une des stratégies énumérées (ou les trois) dans la section précédente et identifier les concepts dans le domaine du problème (à partir des cas d'utilisation) – section 8.1.

2. Exprimer les concepts sous forme de notation UML (diagramme de classes) – section 4.7.

3. Ajouter les associations qui existent entre les concepts. Les associations sont des relations qui lient les différents concepts – section 8.3. Donner les multiplicités aux extrémités des associations.

4. Ajouter les attributs nécessaires à chacun des concepts. L'ensemble des valeurs de ces attributs représente les états des concepts – section 8.4.

Il est parfois difficile de distinguer entre un concept et un attribut. Dans la plupart des cas, un nombre, un texte est normalement associé à un attribut. Dans des cas plus complexes, nous pouvons, dans un premier temps, considérer ces entités comme des concepts. Puis, avec une compréhension accrue des objets de la problématique à résoudre et une relecture des cas d'utilisation, on peut reclasser ces entités en concepts ou en attributs.

Le UML utilise souvent le terme concept pour désigner les objets de la *vraie* vie. Il utilise plutôt le terme **classe** pour identifier les choses réelles. Lorsqu'il traite des classes C++, le UML utilise le terme **classe d'implantation**. Dans la même ligne de pensée, une opération est un service offert par un objet et représente un comportement. Le terme **méthode**, selon le point de vue UML, est la réalisation d'une opération. Le terme **type**, quant à lui, est utilisé par le UML pour désigner une classe qui n'a pas de méthodes. Donc, un type est la spécification d'une classe et non sa réalisation.

8.3 ASSOCIATIONS ENTRE CONCEPTS

Par définition, une association est une relation entre les concepts. Tout comme dans la vie de tous les jours, il peut exister des relations qui n'apportent pas de bénéfice aux parties impliquées. Pour déterminer les associations utiles aux concepts, il faut tenir compte des facteurs suivants :

- Dans le modèle conceptuel, inclure les associations pour lesquelles l'existence de la relation doit être préservée pour une durée de temps (associations de type « besoin-de-connaître »).

- Dans le modèle conceptuel, inclure les associations dérivées de la liste des associations type (présentée ci-dessous).

Si une association envisagée n'est pas cadrée par les facteurs énumérés alors il serait opportun d'examiner l'utilité réelle d'une telle association dans le modèle conceptuel.

8.3.1 ASSOCIATIONS TYPES

Un répertoire des associations type peut aider dans l'obtention des associations du modèle conceptuel. Pour appliquer ce répertoire d'association type, passer en revue les cas d'utilisation et relever systématiquement les associations applicables.

Association	Exemple
A est une partie physique de B	Gène → Chromosome
A est une partie logique de B	Allèle → Gène
A est physiquement contenu dans B	Chromosome → Cellule
A est logiquement contenu dans B	Génotype → individu
A est une description de B	Code génétique → Être humain
A est une caractéristique de B	Cellulite → Inflammation sous-cutané
A est connu de B	Génétique → Humain
A est enregistré par B	Paiement → Guichet automatique
A est un membre de B	Prince → Famille royale
A est une sous-unité (organisationnelle) de B	GPA → ETS
A est utilise par B	Calculatrice → Étudiant
A est géré par B	Emplyé → Employeur
A communique avec B	Professeur → Employeur
A est relié à B	Accident → Ivresse au volant
A effectue des transactions avec B	Utilisateur → Préposé au guichet
A possède B	Étudiant → Calculatrice
Associations propres au domaine de la problématique	

Tableau 17 Répertoire des associations type.

Évidemment, aucun répertoire n'est exhaustif. Il est recommandé d'apporter des ajouts (surtout des associations reliées au domaine du problème) qui peuvent enrichir le contenu du répertoire des associations. Cependant, les associations types :

- A est une partie physique (logique) de B

- A est physiquement (logiquement) contenu dans B

- A est membre de B

sont presques toujours existants dans un modèle conceptuel.

Il peut exister plus d'une association entre deux concepts (voir section 5.5.2). Les associations sont des relations. Elles n'ont pas d'attributs ni d'opérations. Pour des relations complexes, il est recommandé de les transformer en concepts (voir section 5.5.3).

8.3.2 NOM DES ASSOCIATIONS

À ce stade-ci de l'analyse, il est conseillé d'assigner des noms selon le format :

`verbe-préposition`

Le verbe est normalement écrit au temps participe passé suivi de la préposition ou à la 3e personne singulière sans préposition. Voici quelques exemples : `enclenché-par`, `ajuste`, `enregistre`, `contenu-dans`, `payé-par`, `envoyé-à`, etc.

Le but de ce format est de faciliter la lecture. Le rôle des diagrammes UML, à ce stade de l'analyse, est davantage un outil de communication.

8.3.3 ASSOCIATIONS ET VARIABLES MEMBRES

Il existe une relation étroite entre les associations des concepts et les variables membres des classes C++. En effet, les associations sont dans la plupart du temps réalisées sous forme de variables membres (référence, pointeur, etc.). **Cependant**, **lors de l'analyse**, **il est important de ne pas faire cette supposition**. Il est nécessaire de faire l'abstraction de l'aspect implantation des concepts pour ne pas forcer inutilement certaines décisions de conception.

Encore une fois, les activités de la phase d'analyse ont pour but de produire un modèle conceptuel du problème à résoudre. Elles sont nécessaires pour dégager les fonctionnalités du logiciel et les résultats de l'analyse sont des moyens de communication entre les membres de l'équipe de développement.

8.4 ATTRIBUTS DES CONCEPTS

Les attributs sont des données à l'intérieur d'un objet. Les attributs sont donc des informations qu'il faut retenir (mémoriser) par les objets. Ce besoin de mémorisation est normalement indiqué explicitement ou implicitement dans les cas d'utilisation.

Il est recommandé d'employer des attributs simples associés aux types de données dont la représentation est facilement assimilable. Le Tableau 18 donne quelques attributs simples :

Attribut	Exemple
`EnMarche`	Booléen → {Vrai, Faux}
`Couleur`	Entier → {Bleu, Blanc, Rouge}
`Adresse`	Séquence caractères → 256 max
`NombreDeRoue`	Entier → {1, … 4}
`Datation`	Date → xx-xx-xxxx, x ∈ {0, …, 9}
`CodePostal`	Séquence caractères → LCL CLC, L = lettre, C = chiffre
`HeureArrivee`	Temps → {hrs : min : sec}

Tableau 18 Quelques attributs simples.

La spécification des attributs s'effectue selon la convention : `nom_attribut : type_attribut := valeur_initiale` (voir section 5.4.1).

8.4.1 IDENTIFICATION DES ATTRIBUTS

Une façon simple et pratique consiste à classer une donnée comme attribut si son identité n'est pas un facteur significatif dans le contexte de la problématique. Autrement dit, les instances de cette donnée n'ont pas à être identifiées dans la problématique. Par exemple dans un système d'ascenseurs il peut s'avérer que l'identité des étages parcourus par les ascenseurs n'est pas importante dans le système. Ainsi de cet exemple, nous avons :

- Les différentes instances de l'étage « Sous-sol 2 » n'ont pas d'importance dans la problématique. (*Étage* est alors un attribut).
- Les différentes instances de la vitesse de déplacement d'un ascenseur n'ont pas d'importance dans la problématique. (*Vitesse* est alors un attribut).

Par contre, différents ascenseurs peuvent déservir différentes sections de l'immeuble. Dans ce contexte, *Ascenseur* n'est pas un attribut puisque l'identité des ascenseurs est importante dans la problématique.

8.5 COMPORTEMENT DU SYSTÈME

Cette tâche de l'analyse consiste à identifier les événements et les opérations du système logiciel. Pour réaliser cette identification, nous devons construire les diagrammes d'interactions. Ces diagrammes d'interaction nous permettent d'identifier facilement les opérations des concepts, complétant ainsi le modèle conceptuel de la problématique.

Dans le UML, les interactions sont exprimées par des diagrammes de collaboration et de séquence. La discussion présentée dans cette section s'appuiera sur les diagrammes de séquence puisque les deux types de diagrammes (collaboration et de séquence) représentent les mêmes informations.

Un diagramme de séquence sert à illustrer le déroulement des événements d'un cas d'utilisation. Le point de départ est donc la lecture et la compréhension du cas d'utilisation en question. Voici les étapes de construction :

1. Effectuer une lecture attentive du cas d'utilisation.

2. Placer les acteurs et les concepts impliqués dans le diagramme. Les concepts sont ceux déjà dégagés dans les tâches précédentes de l'analyse. Il ne faut pas inventer de nouveaux concepts uniquement dans le but de construire le diagramme.

3. Dessiner des lignes pointillées verticales émanant des acteurs et des concepts.

4. Identifier dans la description du cas d'utilisation les événements enclenchés par les acteurs. Les illustrer dans le diagramme.

5. Identifier dans la description du cas d'utilisation les réponses du système face aux événements enclenchés par les acteurs. Exprimer également les événements générés entre les concepts du cas d'utilisation.

6. Ajouter, si désiré, le texte de la description du cas d'utilisation dans la marge du diagramme.

Voici un exemple d'un diagramme de séquence impliquant un acteur et trois concepts. Dans ce diagramme, l'événement Acheter(x) est enclenché par Acheteur, l'unique acteur de ce diagramme. Les autres événements du diagramme sont des réponses du système et des concepts.

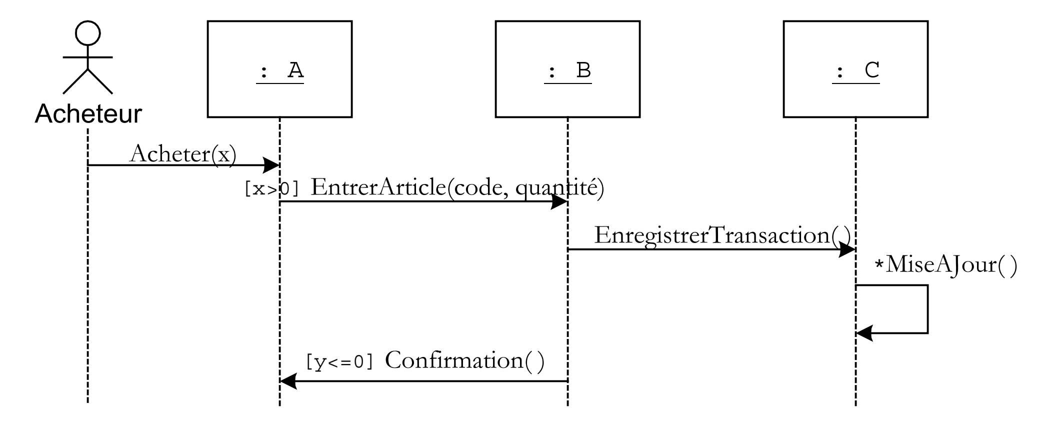


Figure 87 Exemple d'un diagramme de séquence comportant des événements et des réponses.

Noter bien la manière dont les événements sont identifiés. Cette écriture ressemble beaucoup à un appel de fonction et facilite l'interprétation de la signification des événements.

Rappelons-nous encore une fois que les diagrammes d'interactions (collaboration et de séquence) sont construits à partir des cas d'utilisation. Donc, il doit y avoir au

moins un diagramme par cas d'utilisation. De plus, on ne doit pas inventer de nouveaux concepts lors de la création de ces diagrammes. S'il s'avère nécessaire d'ajouter de nouveaux concepts cela signifie que le modèle conceptuel est incomplet. On doit avoir apporté des modifications dans la liste des concepts dégagés.

8.5.1 CRÉATION DES CONTRATS D'OPÉRATIONS

Les opérations sont des services offerts par des concepts. Elles décrivent le comportement des concepts. Nous utilisons les diagrammes d'interactions pour identifier les opérations de chacun des concepts. Comme son nom l'indique, un contrat d'opérations est un document décrivant ce que doivent faire les opérations. Attention, il s'agit ici de définir ce que font les opérations et non comment ces opérations sont implantées.

Voici les étapes impliquées dans l'identification et la création des contrats d'opérations :

1. Identifier les opérations contenues dans les diagrammes d'interactions. Les opérations sont les événements envoyés et reçus des concepts.

2. Créer un contrat pour chacune des opérations.

3. Assigner les opérations dans les concepts dégagés. Normalement, une opération est assignée à un concept identifié par la direction de la ligne fléchée.

Chaque opération est identifiée par une description semblable à celle donnée dans le tableau .

Information	Description
Nom	Nom de l'opération avec la signature complète.
Responsabilité	Une description informelle des tâches à accomplir par cette opération.
Référence	Les numéros de diagrammes UML où figure cette opération.
Note	Indiquer la nature des paramètres de l'opération, les algorithmes envisagés, etc.
Exception	Indiquer les situations anormales possibles.
Sortie/Affichage	Les sorties produites par l'opération.
Pré-conditions	L'état du système avant l'exécution de l'opération.
Post-conditions	L'état du système après l'exécution de l'opération. Utiliser les catégories suivantes pour décrire les post-conditions : – Création ou destruction d'instance de concepts – Modification des attributs – Formation ou élimination d'associations

Tableau 19 Format d'un contrat d'opération.

Un exemple de contrat est donné dans le Tableau 20. Ce contrat identifie l'opération EntrerArticle apparaissant dans le diagramme de séquence de la Figure 87.

Information	Description
Nom	EntrerArticle(Code : Séquence caractères, quantité : Entier)
Responsabilité	Entrer la vente d'un article et l'enregistrer comme une vente.
Référence	Cas d'utilisation : CU12, CU13. Diagramme de séquence : DS12, DS13.
Note	Placer dans une liste temporaire.
Exception	L'article doit être en stock.
Sortie/Affichage	Non applicable.
Pré-conditions	Le code est connu du système. La quantité suffisante en inventaire.
Post-conditions	– Associer l'article dans la catégorie « vente en magasin ». – Soustraire la quantité de l'inventaire.

Tableau 20 Exemple d'un contrat d'opération.

Une attention particulière doit être apportée à l'écriture des post-conditions. En effet, il est très important de bien indiquer la formation ou l'élimination des associations provoquées par l'exécution de l'opération. Par exemple, après l'exécution de EntrerArticle, l'article en question est considéré comme vendu en magasin. Il y a donc formation d'une association entre « article » et « vente en magasin ». Ces actions sont implicites dans l'opération et doivent être décrites d'une manière explicite.

8.6 TÂCHES IMPORTANTES DE L'ANALYSE

Le cheminement présenté dans les sections 8.1 à 8.5 ont pour but de créer rapidement un modèle conceptuel. Cependant, il existe un ensemble de tâches connexes qui sont également nécessaires afin de produire un modèle conceptuel qui est robuste et facile d'entretien. Ces tâches sont présentées ci-dessous :

- Établir les relations entre les cas d'utilisation (voir section 5.6.3).
- Accorder une priorité aux cas d'utilisation. Le processus d'analyse débutera avec les cas les plus prioritaires.
- Étendre le modèle conceptuel en appliquant la généralisation aux concepts dégagés.
- Étendre le modèle conceptuel en appliquant l'agrégation et la composition aux concepts dégagés.
- Approfondir la compréhension du système en créant un diagramme d'état pour chacun des cas d'utilisation. Ces diagrammes d'état seront utilisés lors de la conception et lors de la construction du logiciel pour assurer le bon fonctionnement du logiciel.

  En effet, le concepteur et les codeurs sont guidés par ces diagrammes d'état pour implanter l'ordre des événements qui se déroulent dans le logiciel.

- Rassembler les cas d'utilisation et les diagrammes dans des paquets UML. Relier ces paquets selon les recommandations de la section 5.3.

Enfin, les paquets UML et de leur contenu constituent l'ensemble des résultats de l'analyse. Ces résultats seront utilisés dans la phase de conception qui cherchera à concrétiser le modèle conceptuel dégagé.

8.7 EXEMPLE D'APPLICATION

Nous reprenons l'exemple du système d'ascenseurs de la section 4.8. Nous débutons l'analyse par l'identification des concepts contenus dans les cas d'utilisation (présentés dans la section 5.10.1). Pour cet exemple, nous utiliserons la technique d'identification par catégories avec responsabilités.

8.7.1 IDENTIFICATION DES CONCEPTS

Voici les concepts relevés des cas d'utilisation. Il s'agit d'une première ébauche. Par la suite, nous verrons le raffinement des concepts.

Catégorie de concepts	Concept	Responsabilité
Lieu physique	Étage	Identifie la position des ascenseurs et des passagers.
	Campus	Regroupe les étages de l'École.
Objet physique	Bouton Bi	Permet l'activation des ascenseurs. Il indique la sélection des étages par les passagers. Les boutons Bi sont situés à l'intérieur des ascenseurs.
	Bouton Fi	Permet l'activation des ascenseurs. Il indique la sélection des étages et des directions (HAUT, BAS) par les passagers. Les boutons Fi sont situés sur les étages du campus.
	Lumière témoin	Montre d'une manière visuelle la sélection des boutons Bi et Fi.
	Ascenseur	Représente l'appareil électromécanique qui transporte les passagers d'un étage à l'autre. Il communique avec le contrôleur et maintient une liste de requêtes sur les étages de destination. Il existe trois ascenseurs dans l'École. Ces ascenseurs ne parcourent pas les mêmes étages.
	Panneau	Maintient une liste de boutons. Grâce à cette liste, le panneau est en mesure de fournir la liste des requêtes au moment opportun.
Acteur	Passager	Représente les passagers du système d'ascenseur. Ce sont ces passagers qui sélectionnent les étages de destination des ascenseurs.
Représentation	Liste de boutons	Regroupe les boutons Bi et Fi. La liste de boutons est maintenue et manipulée par le panneau de l'ascenseur.
	Statut de l'ascenseur	Contient l'ensemble des états de fonctionnement des ascenseurs. Il appartient donc aux ascenseurs et est manipulé par ces derniers..
Mécanisme	Contrôleur	Gère le déplacement des ascenseurs. Effectue l'arrêt des ascenseurs en cas de défaillance. Le même contrôleur commande les trois ascenseurs de l'École.

Agrégat d'informations	Statut DONNÉES DÉCISION	Repésente le statut de fonctionnement d'un ascenseur. L'ascenseur présente différents comportements en fonction de la valeur de son statut. Contient les informations nécessaires pour calculer la prochaine destination d'un ascenseur. Contient les informations sur la prochaine destination d'un ascenseur.
Règle et politique	Règles de déplacement	Réalise la logique utilisée dans la sélection des étages de parcours des ascenseurs. Le contrôleur doit consulter les règles de déplacement pour connaître la prochaine destination des ascenseurs.

Les concepts présentés dans le tableau ci-dessus ne sont pas tous utiles. En effet, notre objectif est de réaliser un logiciel capable de simuler le fonctionnement d'un système d'ascenseur. Or, les concepts d'étages, de campus, lumière témoins et de passager ne contribuent pas réellement à solution de la problématique. Voici les constatations importantes :

- Les étages, bien qu'utiles dans l'identification des parcours, n'ont pas de responsabilité réelle dans le système. Nous les transformerons en attributs.
- Le campus de l'École est une entité physique qui sert à regrouper les étages. Or, les étages seront transformés en attributs, le concept campus n'a plus de rôle à jouer.
- Le passager peut sembler important dans la problématique. Après tout, ce sont ces passagers qui sélectionnent les étages de destination. Cependant, la sélection des étages peut être remplacée par une fonction aléatoire. Il n'est donc pas nécessaire de recourir au concept passager pour activer les boutons Bi et Fi du système d'ascenseurs.
- Les lumières témoins servent à indiquer visuellement l'état des boutons (activé, désactivé). Cette responsabilité n'a pas de valeur pratique dans notre contexte. Nous les transformerons en attributs.

Ainsi les concepts retenus sont ceux présentés dans le Tableau 21. À noter que le nom final de ces concepts est indiqué entre parenthèses dans la 2e colonne.

Catégorie de concepts	Concept	Responsabilité
~~Lieu physique~~	~~Étage~~ ~~Campus~~	~~Identifie la position des ascenseurs et des passagers.~~ ~~Regroupe les étages de l'École.~~
Objet physique	Bouton Bi (`Bi`) Bouton Fi (`Fi`) ~~Lumière témoin~~	Permet l'activation des ascenseurs. Il indique la sélection des étages par les passagers. Les boutons Bi sont situés à l'intérieur des ascenseurs. Permet l'activation des ascenseurs. Il indique la sélection des étages et des directions (HAUT, BAS) par les passagers. Les boutons Fi sont situés sur les étages du campus. ~~Montre d'une manière visuelle la sélection des boutons Bi et Fi.~~

	Ascenseur (`Ascenseur`) Panneau (`Panneau`)	Représente l'appareil électromécanique qui transporte les passagers d'un étage à l'autre. Il communique avec le contrôleur et maintient une liste de requêtes sur les étages de destination. Il existe trois ascenseurs dans l'École. Ces ascenseurs ne parcourent pas les mêmes étages. Maintient une liste de boutons. Grâce à cette liste, le panneau est en mesure de fournir la liste des requêtes au moment opportun.
~~Acteur~~	~~Passager~~	~~Représente les passagers du système d'ascenseur. Ce sont ces passagers qui sélectionnent les étages de destination des ascenseurs.~~
Représentation	Liste de boutons (`ListeBtn`) Statut de l'ascenseur (`Statut`)	Regroupe les boutons Bi et Fi. La liste de boutons est maintenue et manipulée par le panneau de l'ascenseur. Contient l'ensemble des états de fonctionnement des ascenseurs. Il appartient donc aux ascenseurs et est manipulé par ces derniers..
Mécanisme	Contrôleur (`Contrôleur`)	Gère le déplacement des ascenseurs. Effectue l'arrêt des ascenseurs en cas de défaillance. Le même contrôleur commande les trois ascenseurs de l'École.
Règle et politique	Règles de déplacement (`Règle`)	Réalise la logique utilisée dans la sélection des étages de parcours des ascenseurs. Le contrôleur doit consulter les règles de déplacement pour connaître la prochaine destination des ascenseurs.

Tableau 21 Concepts retenus pour l'exemple d'application.

8.7.2 IDENTIFICATION DES ASSOCIATIONS

Nous allons établir les associations entre les concepts énumérés dans le Tableau 21. La démarche préconsiée consiste à identifier les associations type (voir section 8.3.1) applicables à notre problématique.

- Les boutons `Bi` et `Fi` héritent de la surclasse `Bouton`.

  L'idée est que `Bi` et `Fi` sont semblables mais possèdent un sous ensemble de caractéristiques différentes. Les boutons `Bi` n'ont pas de direction associée tandis que Fi dispose d'une direction (HAUT et BAS).

- Les boutons `Bi` et `Fi` sont en agrégation dans `ListeBtn`.

  Les boutons `Fi` sont situés sur les étages. Ils appartiennent donc aux étages. Mais les étages ne sont pas des concepts dans notre problématique. Par conséquent, on les met en agrégation dans `ListeBtn`. Les boutons `Bi` appartiennent aux ascenseurs. Cependant, pour simplifier le travail, on a opté pour une relation d'agrégation plutôt qu'une relation de composition avec `ListeBtn`. Il en résulte donc une certaine symétrie entre les boutons `Bi` et `Fi`. Il existe N boutons `Bi` et `Fi` pour chaque `ListeBtn`.

- Le concept `ListeBtn` est en composition avec `Panneau`.

La durée de vie de ListeBtn est identique à celle de Panneau. En effet, ListeBtn n'est une simple structure de données manipulée par le Panneau.

- Le Panneau est en composition avec Ascenseur.

  La durée de vie de Panneau est identique à celle de l'Ascenseur.

- Le concept Ascenseur est en dépendance avec le Statut.

  La dépendance signifie ici que le concept Statut vient modifier le comportement de l'ascenseur. Autrement dit, Statut sera utilisé comme attribut et paramètre d'entrée des opérations de l'Ascenseur. Cette dépendance sera nommée Utilise. Ainsi, Ascenseur *utilise* Statut.

- L'Ascenseur hérite de la surclasse ASC.

  Les ascenseurs partagent un ensemble de caractéristiques communes. Ils diffèrent seulement dans leur identification (A1, A2, A3) et dans leur trajet de parcours.

- Le contrôleur est en relation d'association avec l'Ascenseur.

  Cette relation est bidirectionnelle. Le contrôleur impose l'étage de destination aux ascenseurs en réponse à leur requête. Par contre, c'est bien le contrôleur qui gère les ascenseurs. Donc, cette relation d'association sera nommée Gère. Ainsi, le contrôleur *gère* les **trois** ascenseurs du système.

- L'étage de destination est calculé par application des règles de déplacement.

  Le déplacement des ascenseurs (étage, direction) est décidé par le contrôleur. Ce dernier consulte le concept Règle pour connaître la valeur du déplacement. Donc, le Contrôleur consulte Règle pour décider sur le déplacement (étage, direction) des ascenseurs qui en font la demande.

8.7.3 IDENTIFICATION DES ATTRIBUTS

Par la lecture des cas d'utilisation (voir section 5.10.1), nous pouvons relever les attribtus suivants.

Concept	Attributs
Bouton	état : ETAT où ETAT ∈ {activé, désactivé, hors_ligne} étage : ETAGE où ETAGE ∈ {S2, S1, RC, E1, M1, E2, M2, E3, M3} direction : DIRECTION où DIRECTION ∈ {HAUT, BAS, HAUT_BAS} lumière1 : Booléen
Bi	*Pas d'attributs propres*
Fi	lumière2 : Booléen (pour Fi, lumière1 est pour la direction HAUT et lumière2 est pour la direction BAS).
ListeBtn	boutonBi : list<Bouton> boutonFi : list<Bouton> Nb_BoutonActif : Entier := 0 // nb. de boutons activés

	état : ETAT
Panneau	boutons : ListeBtn état : ETAT Nb_Selection : Entier := 0 // nb. de sélections effectuées
ASC	statut : STATUT := REPOS où STATUT ∊ {REPOS, ARRÊT, EN_MARCHE, DÉFAILLANCE} étage_courant : ETAGE := RC etage_destination : ETAGE := RC direction_courante : DIRECTION := HAUT panneau : Panneau contrôleur : Contrôleur Nb_ChangeDir : Entier := 0 // changements de direction Nb_ÉtageParcourus : Entier := 0 Nb_Arrêt : Entier := 0 // Nb_Repos : Entier := 0 // changements de direction portes : PORTES := FERMÉES où PORTES ∊ {OUVERTES, FERMÉES} id : Entier
Ascenseur	parcours : list<ETAGE> // liste des étages desservis
Contrôleur	ascenseurs : vector<ASC> défaillance : booléen := FAUX règle : Règle état_traitement : vector<TRAITEMENT> où TRAITEMENT ∊ {SOLLICITATION_REÇUE, DEMANDE_INFO, CONSULTE_RÈGLE, COMMANDE, REQUÊTE_TRAITÉE}
Règle	numéro_asc : Entier requêtes : list<ETAGE, DIRECTION> direction_courante : DIRECTION prochain_etage : ETAGE
Statut	statut_courant : STATUT statut_précédent : STATUT
DONNÉES	étage_courant : ETAGE direction_courante : DIRECTION liste_sel : list<ETAGE, DIRECTION>
DÉCISION	étage_destination : ETAGE direction : DIRECTION

Tableau 22 Attributs des classes de l'exemple d'application.

Les attributs du Tableau 22 comprennent un nombre d'énumération. Ainsi, les types ETAT, ETAGE, DIRECTION, STATUT, PORTES et TRAITEMENT sont des énumérations. Nous représenterons les types énumérés par des classes sans opérations. Nous avons utilisé des classes collection pour représenter des attributs de cardinalité supérieure à 1. Par conséquent, parcours : list<ETAGE > signifie que l'attribut parcours est une liste d'étages avec les directions correspondantes. Enfin, les attributs de cardinalité supérieure à 1 mais dont la taille est déterminée sont représentés par des vecteurs d'éléments. Ainsi, ascenseurs : vector<ASC> signifie que l'attribut ascenseurs comprend un nombre déterminé d'objets de type ASC.

L'attribut Nb_BoutonActif de la classe ListeBtn compte le nombre de boutons sélectionnés de l'ascenseur. L'attribut Nb_Selection de la classe Panneau sert à compter le nombre de sélections (des boutons) effectuées par les passagers d'un ascenseur. Enfin les attributs Nb_ChangeDir, Nb_ÉtageParcourus, Nb_Arrêt, et Nb_Repos sont des attributs nécessaires à la comptabilisation des statistiques.

8.7.4 DICTIONNAIRE DES TERMES

Le dictionnaire sert à enregistrer les termes techniques utilisés dans la phase d'analyse. La signification de ces termes techniques peut différer de celle du langage courant. Le dictionnaire des termes agit donc comme un entrepôt de définitions. Ces définitions sont propres au projet en cours. Elles permettent une meilleure interprétation de la documentation par les membres de l'équipe de développement.

Terme	Signification
`Ascenseur`	Lorsque écrit en caractères couriers, le mot `Ascenseur` signifie une classe ou un objet représentant un ascenseur du système d'ascenseurs.
Requête	Selon le point de vue de l'`Ascenseur`, une sélection d'étage par les boutons Bi et Fi effectuée par un passager est considérée comme une requête de l'`Ascenseur`.
Consultation	L'action entreprise par le `Contrôleur` auprès de l'objet `Règle` afin d'obtenir la prochaine destination d'un `Ascenseur`.
`Contrôleur`	Lorsque écrit en caractères couriers, le mot `Contrôleur` signifie une classe ou un objet représentant le contrôleur logique du système d'ascenseurs.
`Règle`	Lorsque écrit en caractères couriers, le mot `Règle` signifie une classe ou un objet représentant les règles de déplacement des ascenseurs.
Sélection	L'action de choisir un étage de destination par les passagers.
Sollicitation	Lorsqu'un `Ascenseur` est au REPOS ou à l'ARRÊT, il doit contacter le contrôleur pour connaître sa prochaine destination. La sollicitation est ce premier contact entre l'`Ascenseur` et le `Contrôleur`.
`Panneau`	Lorsque écrit en caractères couriers, le mot `Panneau` signifie une classe ou un objet représentant le panneau contenant la liste des boutons d'un `Ascenseur`.

Tableau 23 Dictionnaire des termes

8.7.5 DIAGRAMMES DE SÉQUENCE

Le comportement dynamique des objets du modèle conceptuel est capturé par les diagrammes de séquence et de collaboration.

Dans la Figure 88, les passager peuvent effectuer des requêtes en sélectionnant les boutons `Bi` et `Fi`. Les passagers envoient le message `Sélectionner()` aux boutons `Bi`. Ces derniers répondent au message en transmettant `MaJ()` (lire Mise à jour) à la liste des boutons. La liste des boutons tient un compteur interne qui donne le nombre de boutons activés par les passagers. Le message `MaJ()` permet à `ListeBtn` d'effectuer la mise à jour de ce compteur interne. Le message `MaJ()` provoque à son tour la transmission de `Augmenter_compteur()` du `ListeBtn` vers le `Panneau`. Ce dernier tient à jour un compteur qui compte le nombre de sélections enclenchées par les passagers depuis le début de la simulation.

Quand aux boutons `Fi`, le message `Sélectionner(Dir)` est utilisé. Les boutons Fi sont situés sur les étages et disposent d'une direction (HAUT, BAS). Ainsi, la sélection d'un bouton Fi doit accompagner de l'indicateur de direction `Dir`.

Note : Les boutons `Bi`, situés à l'intérieur des ascenseurs, n'ont pas de direction associée. Nous utiliserons la valeur `HAUT_BAS` de l'énumération `DIRECTION` pour indiquer ce fait.

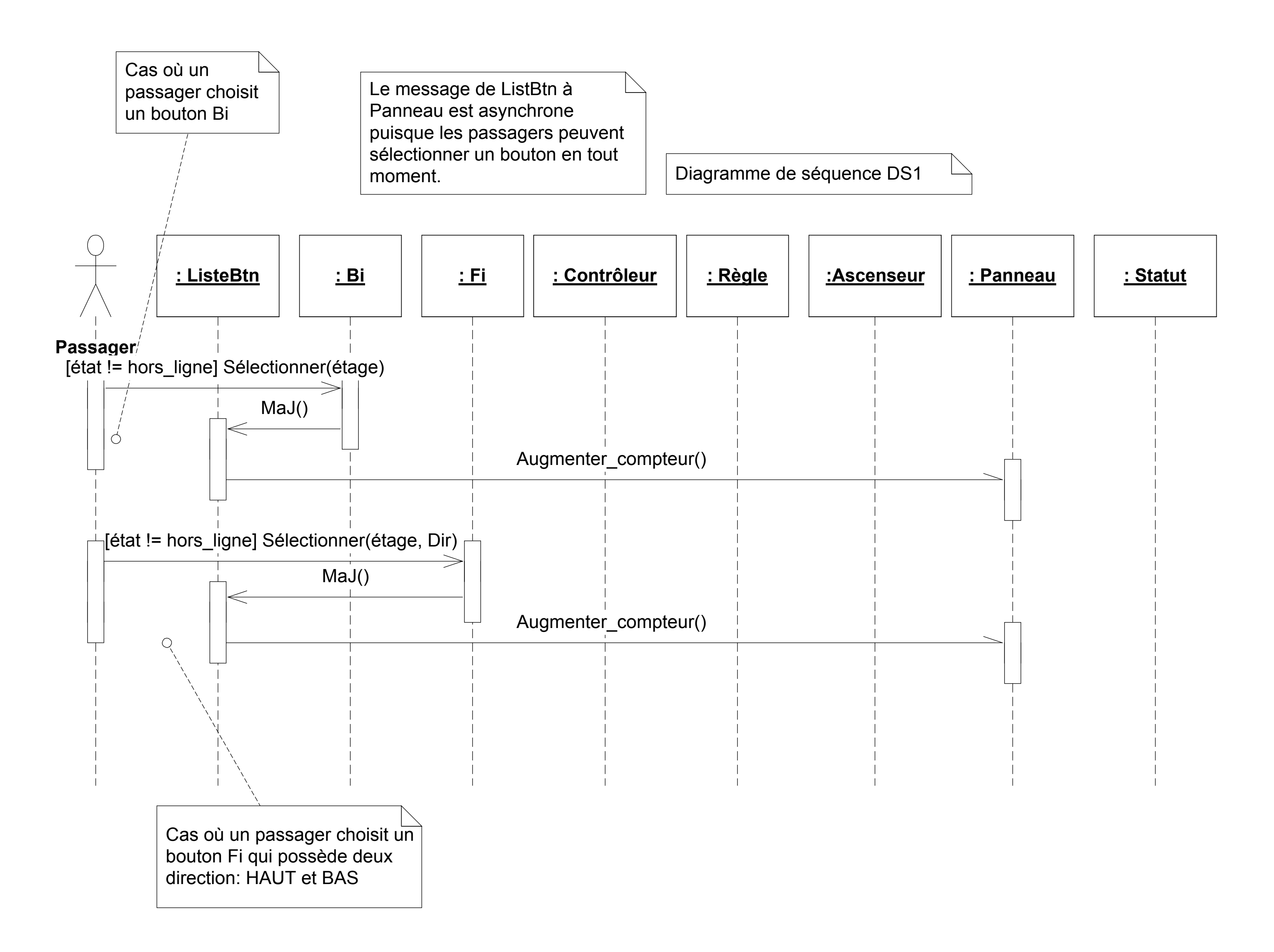


Figure 88 Diagramme de séquence numéro DS1 exprimant la sélection des boutons par les passagers.

À remarquer que le message `Augmenter_compteur()` est asynchrone. L'idée est que le `Panneau` peut être occupé à répondre à une requête de l'`Ascenseur` (voir Figure 89). Le `Panneau` ne peut donc pas répondre immédiatement à ce message venant de `ListeBtn`. De plus, la nature asynchrone de ce message permet à `ListeBtn` et par extension les boutons de continuer immédiatement leur travail sans attendre l'accusé de réception du `Panneau`.

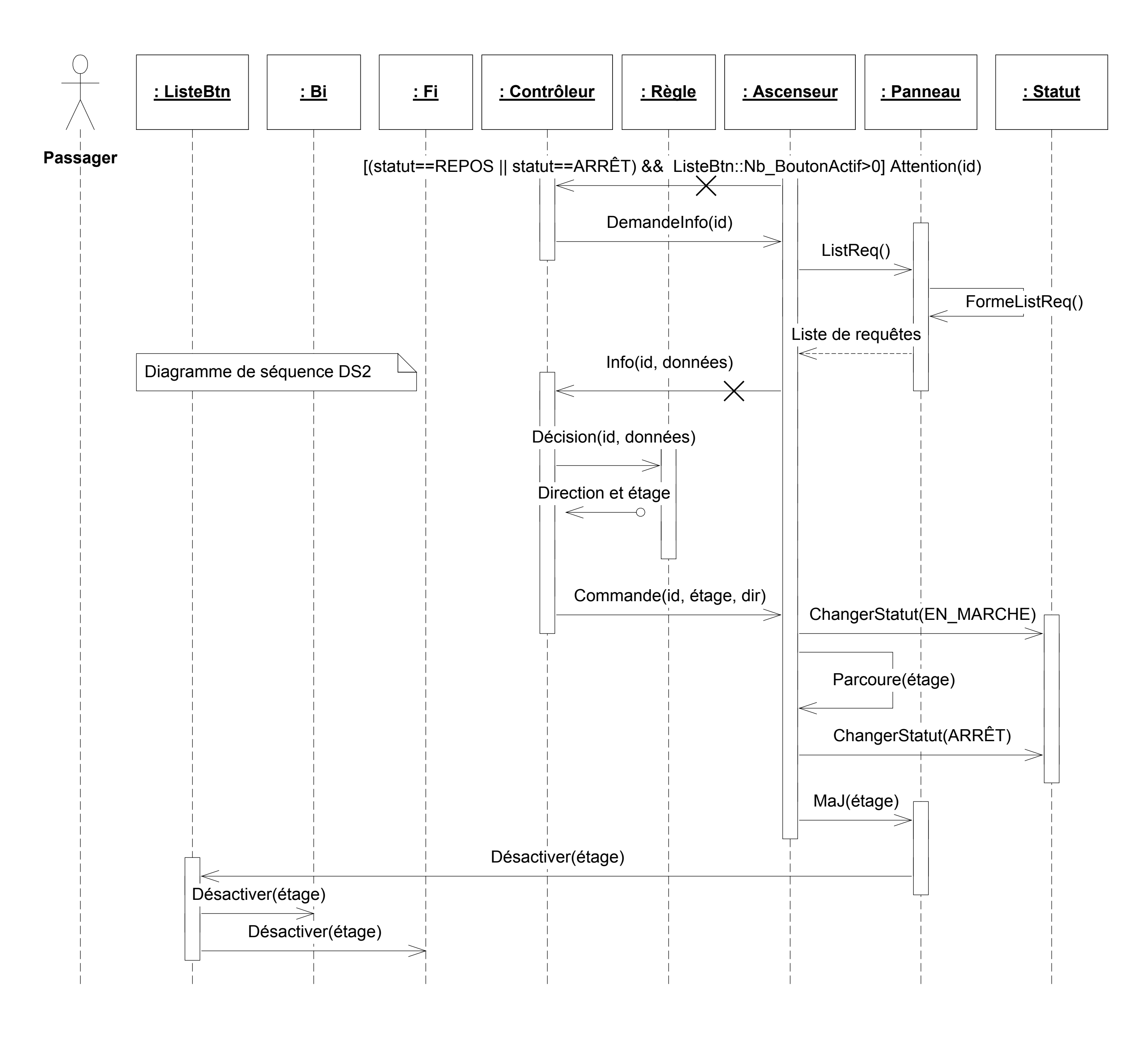


Figure 89 Diagramme de séquence DS2 exprimant les événements impliqués dans l'obtention d'un étage de destination.

Le diagramme de séquence de la Figure 89 débute par un message synchrone `Attention(id)` de l'ascenseur vers le contrôleur lui indiquant que l'ascenseur est au repos (ou arrêté) et qu'il y a au moins une requête indiquée par Nb_BoutonActif (un attribut de `ListeBtn`). Le message `Attention` comporte un paramètre `id` qui identifie l'ascenseur source du message. Le message est de type synchrone puisque l'ascenseur ne peut continuer son travail tant et aussi longtemps que les consignes du contrôleur ne sont pas obtenues. Le contrôleur répond au message `Attention` en envoyant le message `DemandeInfo()` vers l'ascenseur concerné lui demandant fournir son étage courant, sa direction courante et la liste des requêtes en cours. Pour obtenir la liste des requêtes, l'ascenseur passe cette demande à son panneau via le

message `ListReq()`. Le panneau forme la liste des requêtes (cette action est représentée par le message `FormeListReq()`) et la présente à l'ascenseur. À son tour, l'ascenseur retourne les résultats au contrôleur par le biais d'un autre message asynchrone `Info(id, données)`. Le paramètre `id` du message `Info` est important. En effet, plusieurs ascenseurs peuvent être en communication avec le contrôleur, le paramètre `id` permet au contrôleur d'identifier son interlocuteur. Le paramètre `données` est de type `DONNÉES`. Ce type est un concept et renferme les informations pour le calcul de la prochaine destination de l'ascenseur.

Pour décider de l'étage de destination, le contrôleur consulte l'objet des règles de déplacement en invoquant le message `Décision(données)`. L'objet `Règle` détermine la prochaine destination de l'ascenseur à l'aide de l'étage courant et de la liste des requêtes en cours. Après quoi, la commande de déplacement est envoyée à l'ascenseur par le contrôleur via le message `Commande(id, étage, dir)`.

Avant le parcours vers l'étage de destination, l'ascenseur doit changer son statut. Il en est de même à l'arrivée de la destination. L'ascenseur réalise ces tâches par le message `ChangerStatut(stat)`. À l'arrivée l'ascenseur doit alors indiquer à son panneau d'effectuer la mise à jour de son état par le message `MaJ(bouton)`. Puisqu'il s'agit de l'arrivée à la destination de l'ascenseur, la mise à jour consiste à désactiver les boutons `Bi` ou `Fi` sélectionnés à l'aide du message `Désactiver(étage)`.

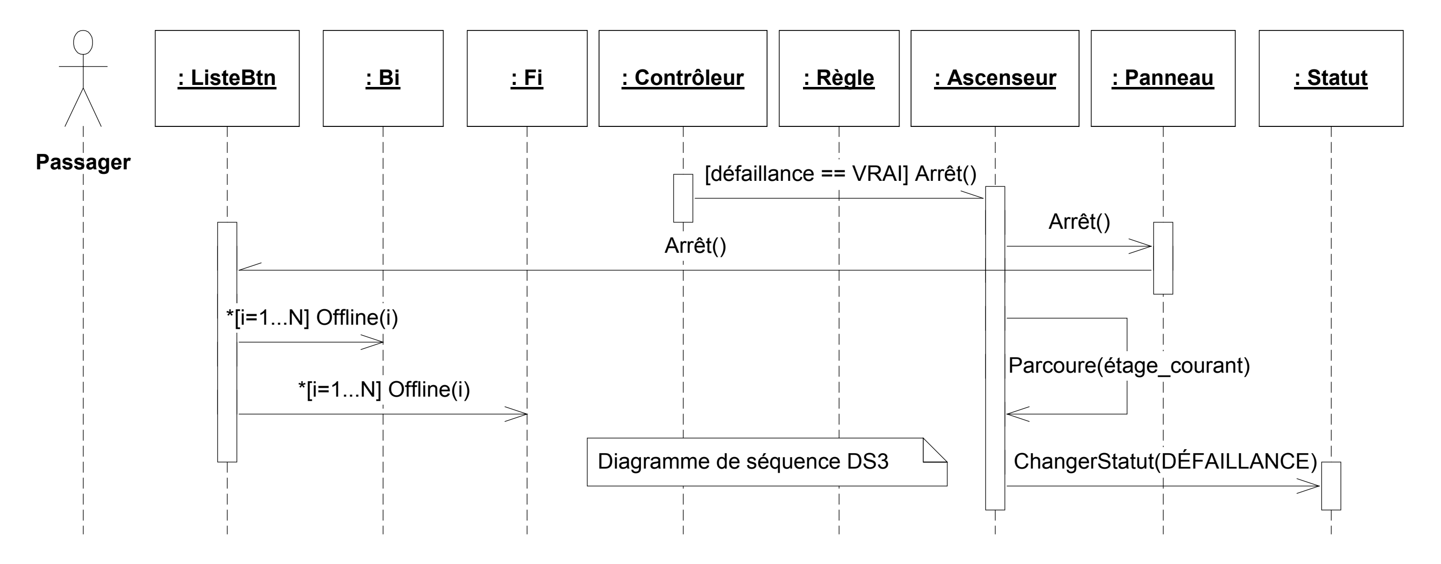


Figure 90 Diagramme de séquence numéro DS3 exprimant les événements associés à l'arrêt d'urgence.

La lecture du diagramme de séquence de la Figure 90 ne doit pas poser de problème. La seule difficulté dans la compréhension de ce diagramme est probablement le message gardé

```
*[i:=1…N] Offline(i)
```

qui exprime une action répétitive. En effet, nous devons comprendre ce message de la façon suivante :

- Le symbole * devant le nom du message signifie l'envoi répétitif du message.

- L'expression gardée devant le nom du message signifie que le message est envoyé si et seulement si l'expression gardée est vraie. Donc, `*[i=1..N]` signifie que le message est envoyé à N reprises puisque l'expression `[i:=1..N]` est vraie pour i = 1 jusqu'à i = N (En informatique, par abuse des définitions, on considère que $i \neq 0 \rightarrow$ VRAI et $i = 0 \rightarrow$ FAUX).

- Ainsi, `*[i:=1..N] Offline(i)` exprime N transmissions du message `Offline(i)`. La raison est qu'il peut exister N boutons associés à l'ascenseur.

8.7.6 CONTRATS DES OPÉRATIONS

À l'aide des diagrammes de séquence, nous pouvons établir les opérations des concepts. Évidemment, nous sommes intéressés par les opérations importantes qui jouent un rôle prépondérant dans le système. Les autres opérations (de support, utilitaires, etc.) seront dégagées lors de la conception proprement dit du système. Voici les contrats résultant des diagrammes de séquence DS1 (Figure 88) à DS3 (Figure 90).

Concept `Bi`

Information	Description
Nom	`Sélectionner(étage : ETAGE) : void`
Responsabilité	Activer le bouton Bi correspondant à l'étage `étage`.
Référence	Cas d'utilisation : CU_DES1. Diagramme de séquence : DS1.
Note	Tous les boutons Bi sont identiques. C'est le passager qui choisit son bouton.
Exception	Il y aura exception lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Lumière témoin allumée.
Pré-conditions	état != hors_ligne, le bouton est en ligne.
Post-conditions	– Pour le bouton sélectionné : état := activé. direction := HAUT_BAS. lumière1 := VRAI, la lumière témoin correspondante est allumée.

Tableau 24 Opération `Sélectionner(étage)` de `Bi`.

Information	Description
Nom	`Désactiver(étage : ETAGE) : void`
Responsabilité	Enlever la sélection du bouton Bi correspondant à l'étage `étage`.
Référence	Cas d'utilisation : CU_DES3. Diagramme de séquence : DS2.

Note	
Exception	Il y aura exception lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Lumière témoin éteinte.
Pré-conditions	état != hors_ligne, le bouton est en ligne.
Post-conditions	– Pour le bouton sélectionné : état := désactivé. lumière1 := FAUX, la lumière témoin correspondante est éteinte.

Tableau 25 Opération `Désactiver(étage)` de `Bi`.

Information	Description
Nom	`Offline(étage : ETAGE) : void`
Responsabilité	Mettre hors circuit le bouton Bi correspondant à l'étage `étage`.
Référence	Cas d'utilisation : CU_DES4. Diagramme de séquence : DS3.
Note	Cette opération est exécutée pour tous les boutons Bi.
Exception	Il y aura exception lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Lumière témoin est éteinte.
Pré-conditions	Aucune
Post-conditions	– Pour le bouton sélectionné : état := hors_ligne. lumière1 := FAUX, la lumière témoin correspondante est éteinte.

Tableau 26 Opération `Offline(étage)` de `Bi`.

Concept `Fi`

Information	Description
Nom	`Sélectionner(étage : ETAGE, Dir : DIRECTION) : void`
Responsabilité	Activer un bouton Fi correspondant à l'étage `étage` en spécifiant la direction de déplacement `dir` (HAUT, BAS).
Référence	Cas d'utilisation : CU_DES2. Diagramme de séquence : DS1.
Note	On n'accepte pas une sélection avec Dir == HAUT_BAS (le passager doit sélectionner HAUT puis sélectionner BAS).
Exception	Dir == HAUT_BAS et lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Lumière témoin associée à la direction Dir allumée.
Pré-conditions	état != hors_ligne, le bouton est en ligne.

Post-conditions	– Pour le bouton sélectionné : état := activé. Régler la direction. Si direction = HAUT && Dir = BAS ou vice versa direction = HAUT_BAS. Autrement direction = Dir. lumière1 := VRAI, si Dir == HAUT. lumière2 := VRAI, si Dir == BAS. lumière1 := lumière2 := vrai, si Dir == HAUT_BAS.

Tableau 27 Opération `Sélectionner(étage, Dir)` de `Fi`.

Information	Description
Nom	`Désactiver(étage) : void`
Responsabilité	Enlever la sélection du bouton Fi correspondant à l'étage `étage`.
Référence	Cas d'utilisation : CU_DES3. Diagramme de séquence : DS2.
Note	Une fois l'ascenseur rendu à la destination, on désactive les lumières témoins des deux directions. Puisque peu importe la direction désirée par le passager, il peut embarquer une fois les portes de l'ascenseur sont ouvertes.
Exception	Lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Les deux lumières témoins éteintes.
Pré-conditions	état != hors_ligne, le bouton est en ligne.
Post-conditions	– Pour le bouton sélectionné : état := désactivé. lumière1 := lumière2 := FAUX, les lumières témoins correspondantes sont éteintes.

Tableau 28 Opération `Désactiver(étage)` de `Fi`.

Information	Description
Nom	`Offline(étage) : void`
Responsabilité	Mettre hors circuit le bouton Fi correspondant à l'étage `étage`.
Référence	Cas d'utilisation : CU_DES4. Diagramme de séquence : DS3.
Note	Cette opération est exécutée pour tous les boutons Bi.
Exception	Lorsque `étage` ∉ `Ascenseur::parcours`. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Les deux lumières témoins sont éteintes.
Pré-conditions	Aucune
Post-conditions	– Pour le bouton sélectionné : état := hors_ligne. lumière1 := lumière2 := FAUX, les lumières témoins correspondantes sont éteintes.

Tableau 29 Opération `Offline(étage)` de `Fi`.

Concept ListeBtn

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	MaJ() : void
Responsabilité	Augmenter de 1 le compteur Nb_BoutonActif. Sert à compter le nombre de boutons activés.
Référence	Cas d'utilisation : CU_DES1. Diagramme de séquence : DS1.
Note	
Exception	Non applicable.
Sortie/Affichage	La valeur de Nb_BoutonActif.
Pré-conditions	Aucune
Post-conditions	– Nb_BoutonActif += 1.

Tableau 30 Opération MaJ() de ListeBtn.

Information	Description
Nom	Arrêt() : void
Responsabilité	Mettre hors ligne tous les boutons Bi et Fi.
Référence	Cas d'utilisation : CU_DES4. Diagramme de séquence : DS3.
Note	Cette opération est enclenchée par le Panneau en réponse à une défaillance du système (défaillance détectée par le Contrôleur). **Cette opération est asynchrone**.
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	Aucune.
Post-conditions	– état := hors_ligne. – Tous les boutons Bi et Fi de l'ascenseur sont en état hors ligne.

Tableau 31 Opération Arrêt() de ListeBtn.

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	Désactiver(étage : ETAGE) : void
Responsabilité	Désactiver la sélection des boutons Bi et Fi identifié par son numéro d'étage. Également, soustraire de 1 le compteur Nb_BoutonActif.
Référence	Cas d'utilisation : CU_DES3. Diagramme de séquence : DS2.
Note	Cette opération provoque l'exécution de l'opétation Désactiver() des boutons Bi et Fi.
Exception	Voir l'exception de des opérations Bi::Désactiver() et Fi::Désactiver().
Sortie/Affichage	La valeur de Nb_BoutonActif.
Pré-conditions	Aucune.

Post-conditions	– Nb_BoutonActif -= 1..

Tableau 32 Opération Désactiver(étage) de ListeBtn.

Concept Panneau

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	Augmenter_compteur() : void
Responsabilité	Augmenter de 1 le compteur Nb_Selection. Sert à compter le nombre de requêtes effectuées par les passagers.
Référence	Cas d'utilisation : CU_DES1. Diagramme de séquence : DS1.
Note	La valeur du comtpeur Nb_Selection incrémente toujours. **Cette opération est asynchrone**.
Exception	Non applicable.
Sortie/Affichage	La valeur de Nb_Selection.
Pré-conditions	Aucune
Post-conditions	– Nb_Selection += 1.

Tableau 33 Opération Augmenter_compteur() de Panneau.

Information	Description
Nom	ListReq() : list<ETAGE, DIRECTION>
Responsabilité	Retourner la liste des requêtes courantes de l'ascenseur.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	Le panneau doit former une liste dynamique contenant les requêtes courantes de l'ascenseur. Cette liste comprend les étages sollicités et la direction de déplacement correspondante.
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	Aucune.
Post-conditions	– La liste list<ETAGE, DIRECTION> est formée dynamiquement contenant les requêtes courantes de l'ascenseur. – Cette liste est retournée comme paramètre de retour.

Tableau 34 Opération ListReq() de Panneau.

Information	Description
Nom	FormeListReq() : list<ETAGE, DIRECTION>
Responsabilité	Parcourir ListeBtn à la recherche de boutons activés. Un bouton activé signifie que l'étage correspondant est sollicité par les passager. On place les étages sollicités ainsi que la direction de déplacement désirée dans une liste dynamique.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	Cette opération est exécutée par Panneau::ListReq().

Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	ListeBtn existe.
Post-conditions	– La liste list<ETAGE, DIRECTION> est formée dynamiquement contenant les requêtes courantes de l'ascenseur. – Cette liste est retournée comme paramètre de retour.

Tableau 35 Opération FormeListReq() de Panneau.

Information	Description
Nom	MaJ(étage : ETAGE) : void
Responsabilité	Signaler l'arrivée de l'ascenseur à un étage de destination. Désactiver les boutons correspondants à l'étage étage.
Référence	Cas d'utilisation : CU_DES3. Diagramme de séquence : DS2.
Note	Cette opération provoquera l'exécution de l'opération ListeBtn::Désactiver(étage).
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	ListeBtn existe.
Post-conditions	– Exécution de ListeBtn::Désactiver(étage).

Tableau 36 Opération MaJ(étage) de Panneau.

Information	Description
Nom	Arrêt() : void
Responsabilité	Une défaillance a été détectée. Mettre les boutons hors ligne.
Référence	Cas d'utilisation : CU_DES4. Diagramme de séquence : DS3.
Note	Cette opération enclenche l'exécution de l'opération ListeBtn::Arrêt().
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	ListeBtn existe.
Post-conditions	– Exécution de ListeBtn::Arrêt().

Tableau 37 Opération Arrêt() de Panneau.

Concept Ascenseur

Information	Description
Nom	DemandeInfo(id : Entier) : void
Responsabilité	Provoquer la formation d'une liste de requêtes courantes pour l'ascenseur identifié par le paramètre id.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.

Note	L'exécution de l'opération DemandeInfo() indique à l'ascenseur que le Contrôleur est prêt à déterminer sa prochaine destination. L'Ascenseur doit retourner une liste contenant les requêtes courantes (étages et direction associée).
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	Aucune
Post-conditions	– La liste des requêtes courantes est créée. – Exécution de l'opération Contrôleur::Info(id, données) où le paramètre données représente la liste des requêtes courantes de l'ascenseur.

Tableau 38 Opération DemandeInfo(id) de Ascenseur.

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	Commande(id : Entier, étage : ETAGE, dir : DIRECTION) : void
Responsabilité	Donner la prochaine destination à l'ascenseur identifié par le paramètre id. Cette destination comprend l'étage et la direction de déplacement.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	
Exception	Non applicable.
Sortie/Affichage	La valeur de Nb_ChangeDir.
Pré-conditions	Aucune.
Post-conditions	– Si direction_courante != dir alors Nb_ChangeDir += 1. – étage_destination := étage. – direction_courante := dir.

Tableau 39 Opération Commande() de Ascenseur.

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	Parcoure(étage : ETAGE) : Booléen
Responsabilité	Déplacer l'ascenseur en utilisant la direction courante jusqu'à l'étage de destination étage. Comptabiliser les étages parcourus.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	Cette opération retourne FAUX tant et aussi longtemps que l'ascenseur n'est pas arrivé à la destination.
Exception	Lorsque étage ∉ Ascenseur::parcours. C'est-à-dire, l'étage sélectionné doit figurer dans la liste des étages desservis par l'Ascenseur.
Sortie/Affichage	Montrer le déplacement de l'ascenseur d'étage en étage.
Pré-conditions	

Post-conditions	– `Nb_ÉtageParcourus +=` Nombre d'étages parcours par l'ascenseur dans cette séquence de déplacement. – étage_courant == étage_destination. – La valeur de retour de l'opération est VRAI.

Tableau 40 Opération `Parcoure(étage)` de `Ascenseur`.

Information	Description
Nom	`Arrêt() : void`
Responsabilité	Une défaillance a été détectée. Mettre hors ligne le panneau des boutons.
Référence	Cas d'utilisation : CU_DES4. Diagramme de séquence : DS3.
Note	
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	`ListeBtn` existe et que la variable `Contrôleur::défaillance == VRAI`.
Post-conditions	– Exécution de `ListeBtn::Arrêt()`.

Tableau 41 Opération `Arrêt()` de `Ascenseur`.

Concept `Statut`

Cette opération effectue également le calcul des statistiques.

Information	Description
Nom	`ChangerStatut(stat : STATUT) : void`
Responsabilité	Changer le statut de l'ascenseur.
Référence	Cas d'utilisation : CU_DES3, CU_DES4, CU_DES5. Diagramme de séquence : DS2, DS3
Note	Les changements de statut sont accompagnés par des changements de comportement de l'ascenseur.
Exception	Lorsque `stat` ∉ `STATUT`. C'est-à-dire, la valeur du paramètre `stat` est invalide.
Sortie/Affichage	Non applicable.
Pré-conditions	Aucune

Post-conditions	– `statut_précédent := statut_courant`
	– `statut_courant := stat`
	– Si `statut_courant == REPOS`,
	`Ascenseur::étage_destination := Ascenseur::étage_courant`
	`Ascenseur::portes := FERMÉES`
	`Nb_Repos += 1`
	– Si `statut_courant == ARRÊT`,
	`Ascenseur::étage_destination := Ascenseur::étage_courant`
	`Ascenseur::portes := OUVERTES`
	`Nb_Arrêt += 1`
	– Si `statut_courant == EN_MARCHE`,
	`Ascenseur::portes := FERMÉES`
	– Si `statut_courant == DÉFAILLANCE`,
	`Ascenseur::étage_destination := Ascenseur::étage_courant`
	`Ascenseur::portes := OUVERTES`

Tableau 42 Opération `ChangerStatut(stat)` de `statut`.

Concept `Contrôleur`

Information	Description
Nom	`Attention(id : Entier) : void`
Responsabilité	Avertir le contrôleur l'arrivée d'une sollicitation venant de l'ascenseur identifié par le paramètre `id`.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2
Note	Cette opération est le début d'une séquence d'échange entre le contrôleur et l'ascenseur. **Cette opération est synchrone**.
Exception	Non applicable.
Sortie/Affichage	Non applicable.
Pré-conditions	(Ascenseur::statut == REPOS) || (Ascenseur::statut == ARRÊT) et qu'il y a au moins 1 sélection effectuée qui n'est pas encore traitée.
Post-conditions	– `état_traitement[id] := SOLLICITATION_REÇUE` – Effectuer l'opération `ascenseurs[id]::DemandeInfo()` dès que possible.

Tableau 43 Opération `Attention(id)` de `Contrôleur`.

Information	Description
Nom	`Info(id : Entier, données : DONNÉES) : void`
Responsabilité	Recevoir les sélections courantes de l'ascenseur (paramètre `données`) identifié par le paramètre `id`.

Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	Cette opération s'inscrit dans la séquence d'échange entre le contrôleur et l'ascenseur. **Cette opération est synchrone**.
Exception	Le paramètre `données` ne doit pas être vide.
Sortie/Affichage	Non applicable.
Pré-conditions	Assembler les informations suivantes dans le paramètre données : – Étage courant de l'ascenseur. – Direction courante de l'ascenseur. – Liste des sélections de l'ascenseur.
Post-conditions	– `état_traitement[id] := DEMANDE_INFO` – Effectuer l'opération `Règle::Décision()` dès que possible.

Tableau 44 Opération `Info(id, données)` de `Contrôleur`.

Concept `Règle`

Information	Description
Nom	`Décision(id, ENTIER, données : DONNÉES) : DECISION`
Responsabilité	Déterminer l'étage et la direction de déplacement en se basant sur les informations du paramètre `données`.
Référence	Cas d'utilisation : CU_DES5. Diagramme de séquence : DS2.
Note	
Exception	Le paramètre `données` ne doit pas être vide.
Sortie/Affichage	Non applicable.
Pré-conditions	
Post-conditions	– `numéro_asc := id` – `requêtes := données.liste_sel` – `direction_courante := données.direction_courante` – L'étage de destination et la direction de parcours sont déterminés.

Tableau 45 Opération `Décision(id, données)` de `Règle`.

8.7.7 DIAGRAMMES D'ACTIVITÉS

Nous complémentons les diagrammes de séquence par des diagrammes d'activités qui explicitent les activités concourantes du système d'ascenseurs.

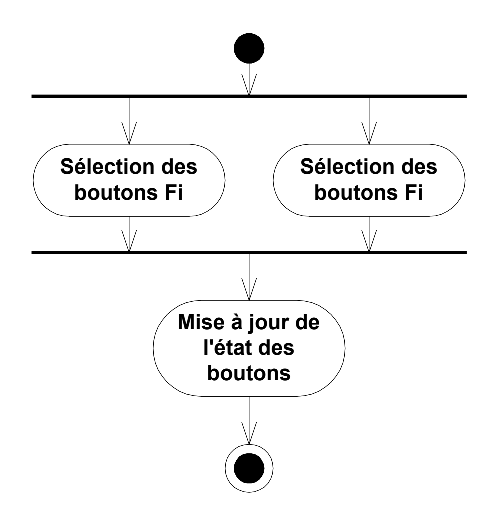


Figure 91 Diagramme d'activités montrant la sélection des boutons d'un ascenseur.

La Figure 91 indique clairement que la sélection des ascenseurs est réalisée par des boutons Bi et/ou par des boutons Fi. Il y aura mise à jour de l'état des boutons peu importe le bouton sélectionné. Dans la Figure 92, les activités concernent une transaction entre l'ascenseur et le contrôleur. Cette transaction est une séquence impliquant la sollication du contrôleur par un ascenseur. Après avoir obtenu les informations nécessaires, le contrôleur consulte les règles de déplacement afin de dicter la prochaine destination. L'ascenseur, en recevant l'étage de destination et la direction de parcours, déplacera automatiquement vers la destination. Il est à noter que cette séquence d'activités est entreprise d'une manière concourante par les ascenseurs. Autrement dit, le contrôleur doit pouvoir accepter plus d'une sollicitations à la fois et dans les plus brefs délais.

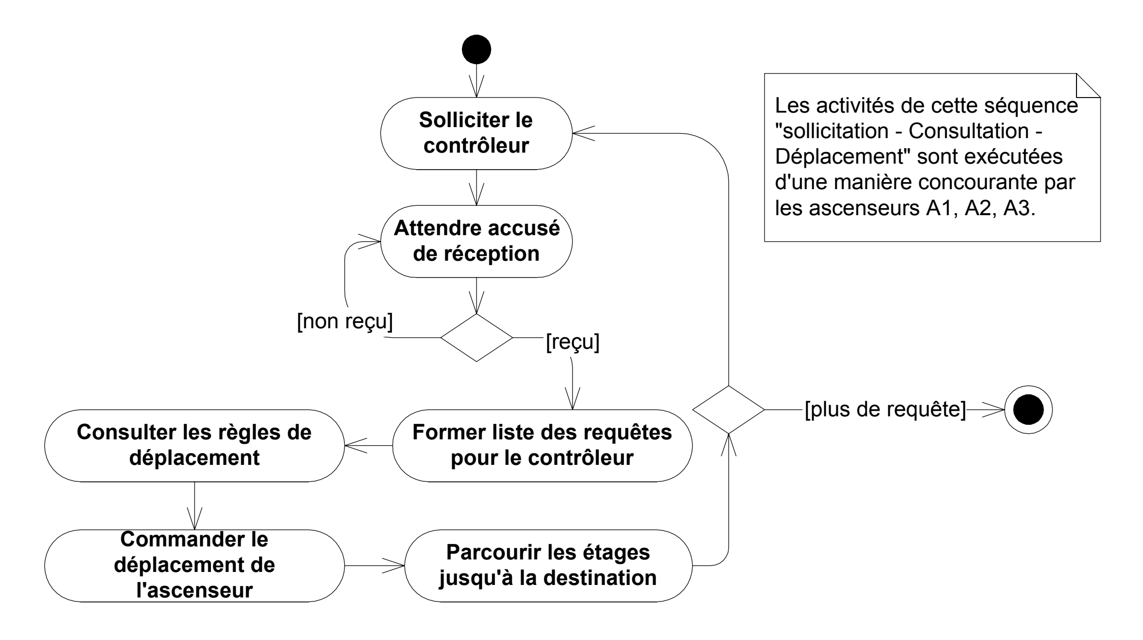


Figure 92 Diagramme d'activités montrant la séquence « sollicitation – consultation – déplacement ».

8.7.8 DIAGRAMMES DE COLLABORATION

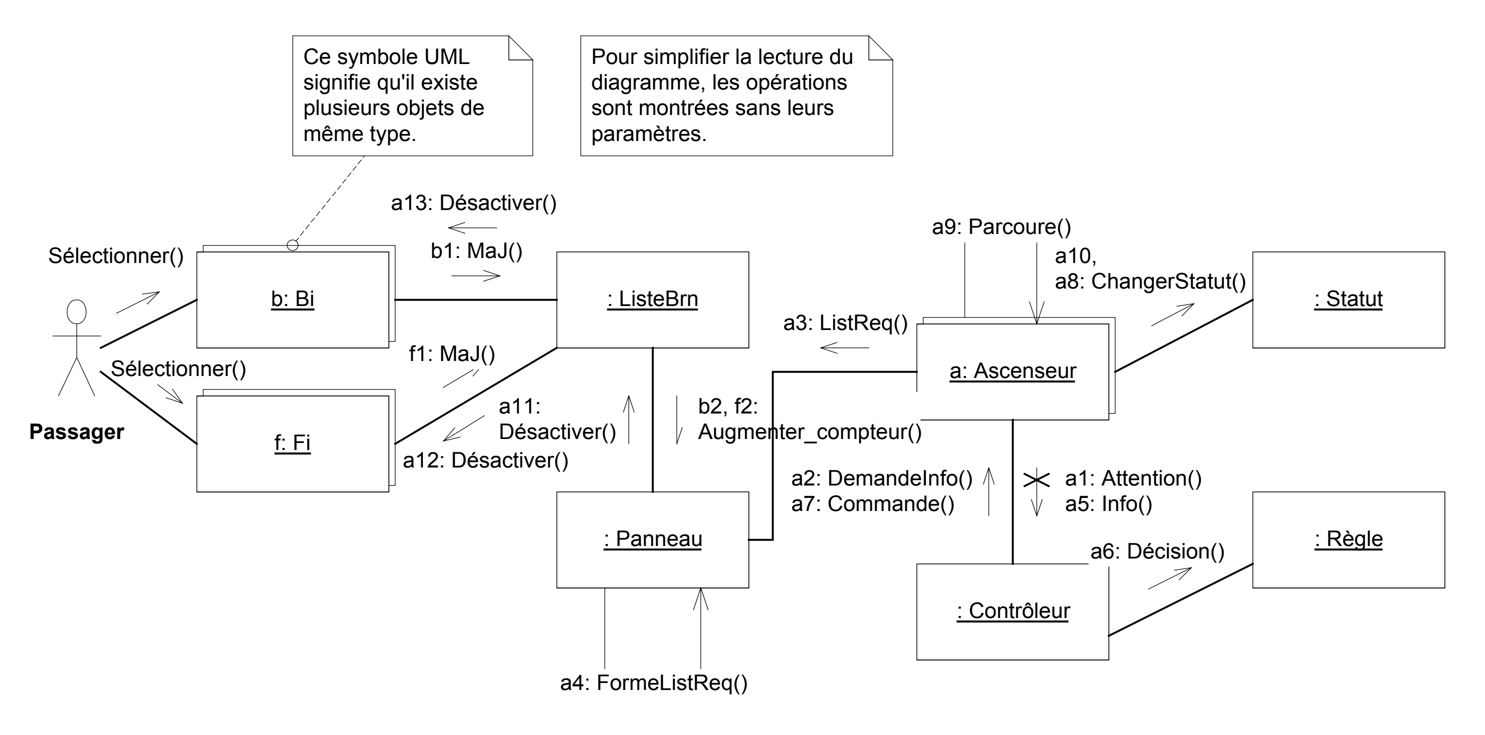


Figure 93 Diagramme de collaboration montrant l'utilisation nornale du système d'ascenseurs.

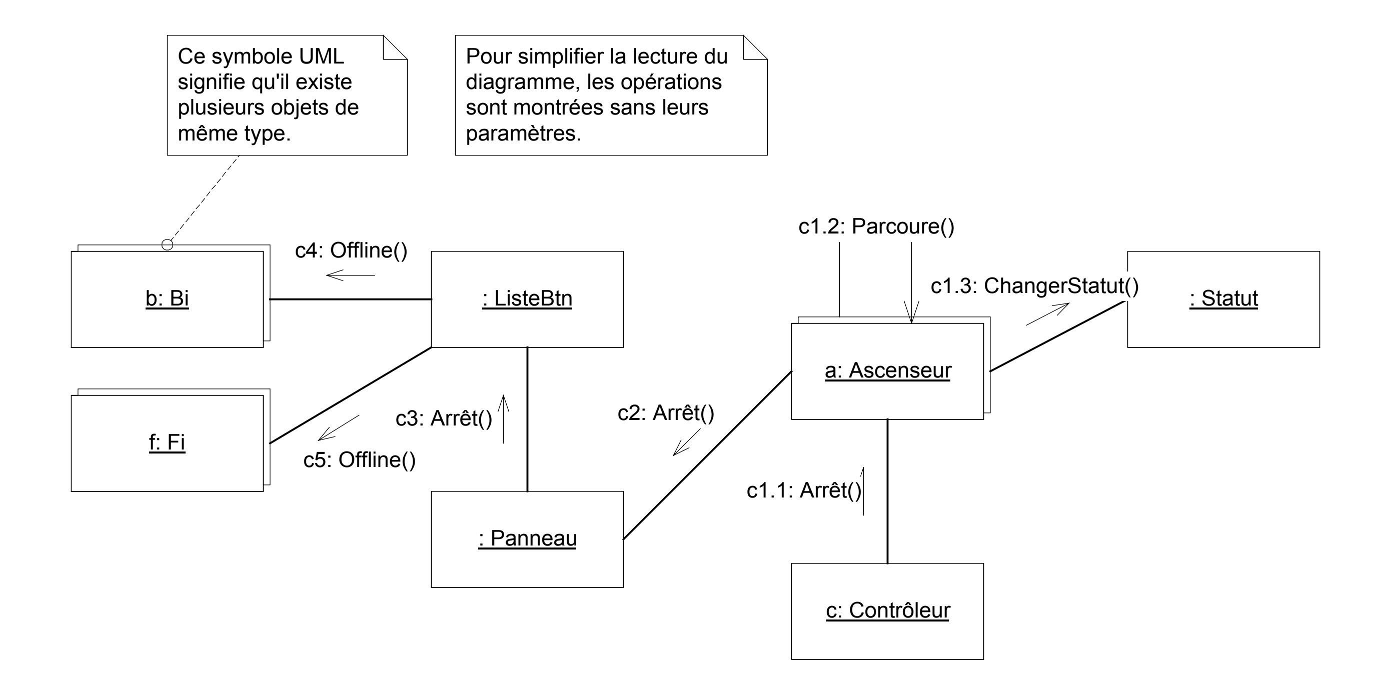


Figure 94 Diagramme de collaboration montrant la situation de défaillance du système d'ascenseurs.

8.7.9 DIAGRAMMES D'OBJETS

Les diagrammes d'objets servent à illustrer les relations entre objets à un moment donné dans le déroulement des activités. Nous pouvons imaginer les diagrammes d'objets comme des photos prises à des moments précis. Ainsi, les objets peuvent avoir des valeurs assignées à leurs attributs. Certaines relations dynamiques entre objets peuvent être établies alors d'autres peuvent ne pas être montrées. Évidemment, les relations statiques entre objets sont toujours illustrées dans ces diagrammes. Ainsi, nous avons à la Figure 95 tous les objets impliqués dans cet exemple d'application : Les trois ascenseurs A1, A2 et A3 (id = 1, id = 2 et id = 3) lié à leur `Panneau`. Le `Panneau` est lié à l'objet `ListeBtn` et ce dernier est lié aux boutons `Bi` et `Fi`. Les trois ascenseurs sont gérés par le `Contrôleur`. L'objet `Règle` est utilisé par le Contrôleur afin de calculer le déplacement des ascenseurs.

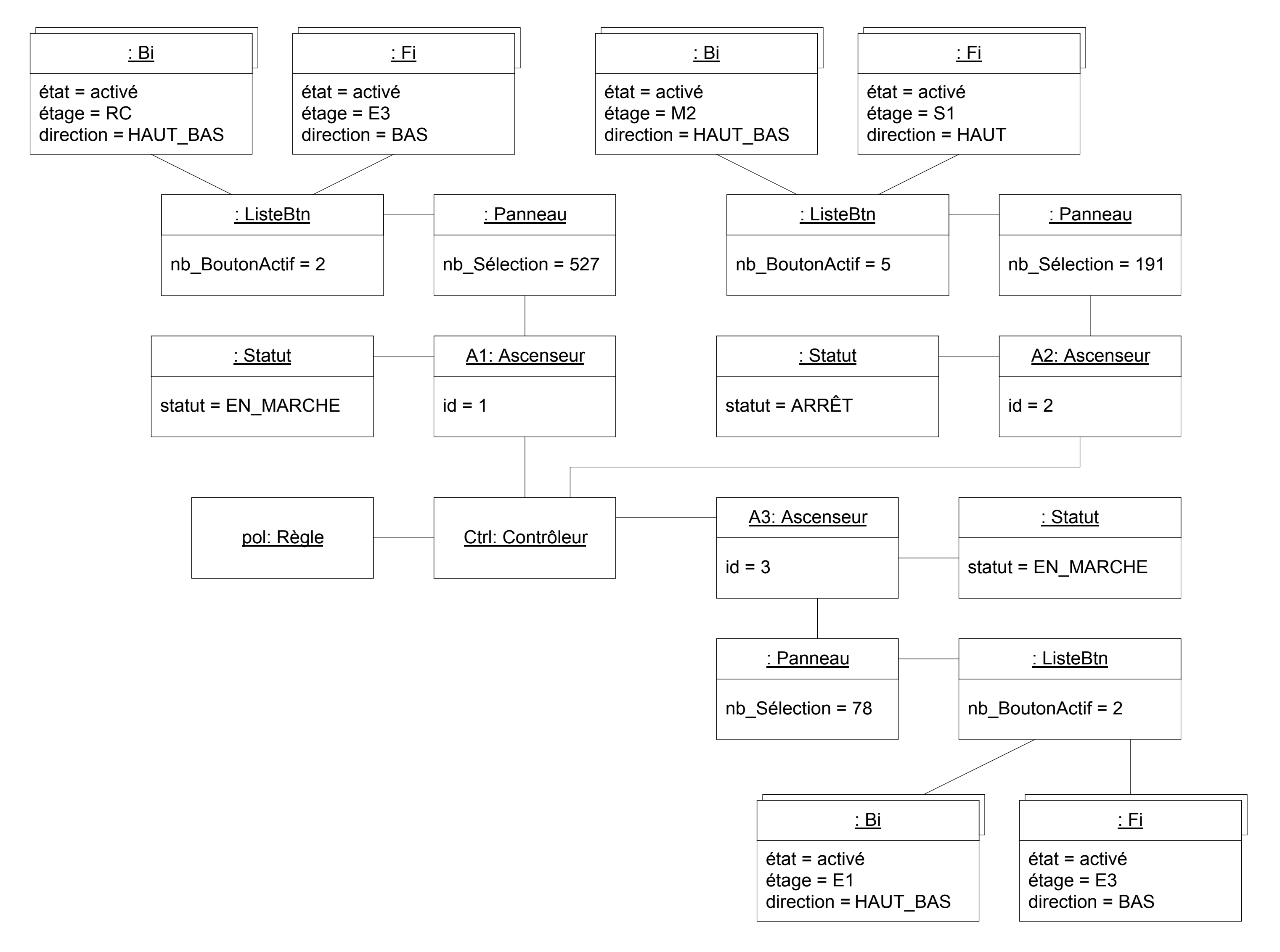


Figure 95 Diagramme d'objets de l'exemple d'application.

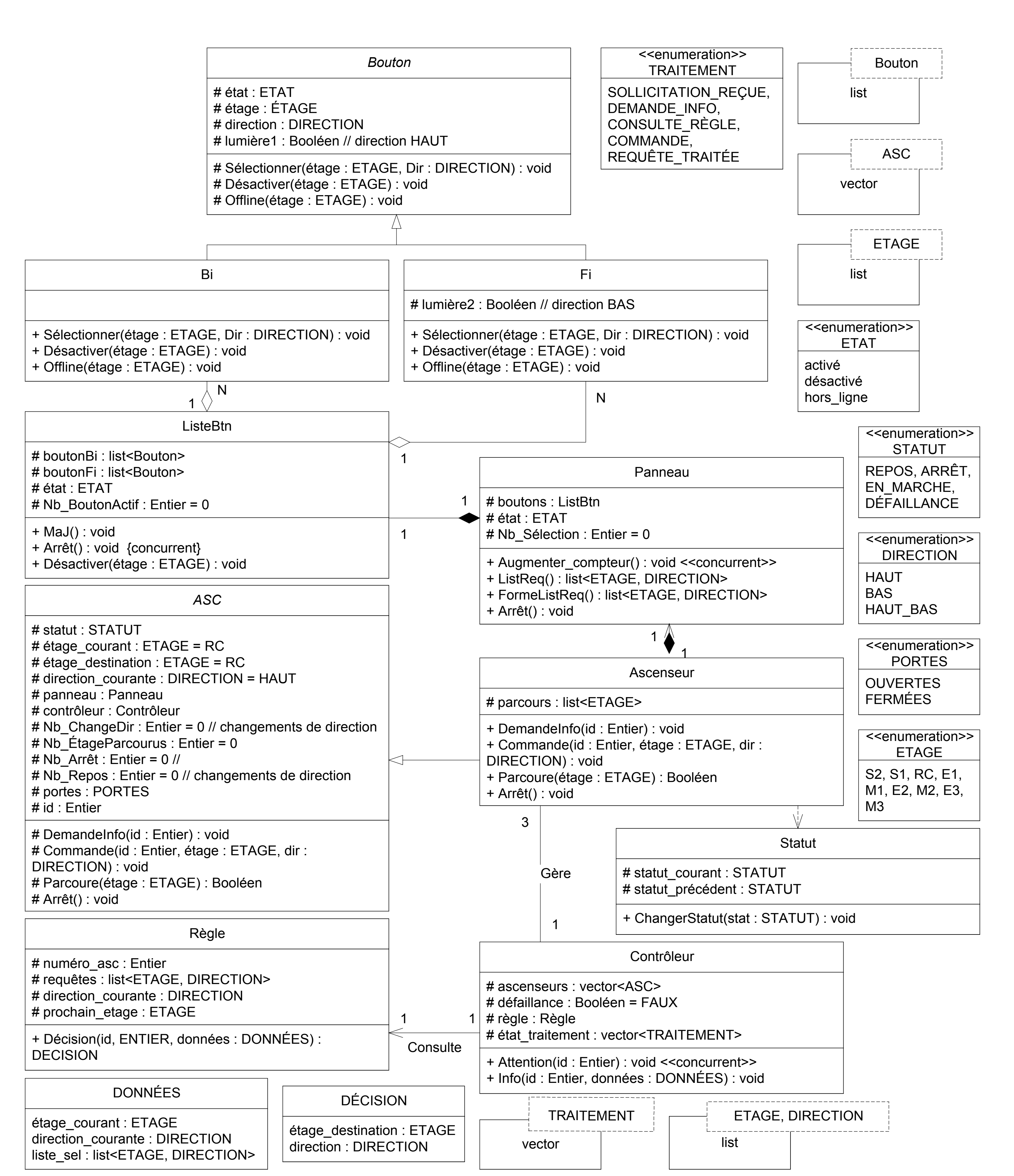


Figure 96 Classes de l'exemple d'application.

LECTURE SUGGÉRÉE

Les références qui ont aidé à la rédaction de ce chapitre sont :

[MART95] Martin, J., Odell, J., *Object-Oriented Methods : A Foundation*, Prentice-Hall, 1995.

[LARM98] Larman, Craig, *Applying UML and Patterns. An introduction to object-oriented analysis and design*, Prentice-Hall, 1998.

[PRES00] Pressman, R.S., *Software Engineering : A practitionner's approach*, McGraw-Hill, 2000.

[MULL97] Muller, Pierre-Alain, *Instant UML*, Wrox Press, 1997.

PROBLÈMES

À venir

CHAPITRE 9

Conception orientée objet

« Il s'agit de votre première rencontre avec une entité non humaine : un extraterrestre (ET). Si vous gérez bien la situation, vous serez le plus grand héros vivant et vous ferez une fortune en vendant votre histoire aux médias. Si vous manquez à votre tâche, la conséquence pourrait être terrible : une guerre interstellaire pouvant anéantir l'humanité … »

— *Jonathan Vos Post, Making Contact.*

La conception orientée objet consiste à dégager un ou des modèles de conception représentant la solution de la problématique. Les concepts dans notre cas sont des classes et leur identification est une tâche très importante de l'analyse orientée objet. Ce chapitre présente les stratégies nécessaires afin de transformer le modèle conceptuel qui explique le problème à résoudre en un modèle de conception qui apporte des solutions au problème formulé.

9. MODÈLES DE CONCEPTION

Le modèle conceptuel dégagé lors de l'analyse sert à expliquer les exigences du logiciel. Dans le contexte de la conception orientée objet, nous devons reprendre le modèle conceptuel et apporter des éléments concrets reliés à la réalisation du logiciel. Mais la tâche la plus importante lors de la conception orientée objet demeure encore l'assignation des responsabilités aux objets.

Le fonctionnement des systèmes orientés objet repose sur les échanges de messages entre objets afin d'accomplir des tâches prescrites. Dans ce sens, les objets interagissent entre eux pour réaliser les fonctionnalités désirées. Ainsi, les objets sont **responsables** des manoeuvres exigées. L'assignation des responsabilités a pour but de bien délimiter le rôle et les capacités des objets afin de former un ensemble d'objets cohérents travaillant en collaboration. Une conception robuste et efficace est donc le résultat d'une assignation judicieuse des responsabilités à travers l'ensemble des objets dégagés.

Pour faciliter le travail, nous utiliserons les patrons de conception disponibles. Ces patrons de conception représentent des assignations de responsabilités applicables à

des situations concrètes. De plus, ces patrons de conception ont été appliqués en pratique par des spécialistes de l'approche orientée objet. Nous présenterons quelques-uns de ces patrons dans les sections subséquentes de ce chapitre.

9.1 TÂCHES PRÉLIMINAIRES

Les tâches préliminaires à la conception sont énumérées ci-dessous :

1. Expliciter la séquence des événements dans les diagrammes d'interactions

   Pour les diagrammes de collaboration, numéroter les messages. Utiliser un système de numérotation capable d'exprimer la simultanéité des messages. De même pour les diagrammes de séquence. Au besoin, recréer des messages plus explicites.

2. Ajouter les messages reliés à la création et à la destruction des objets dans les diagrammes d'interactions.

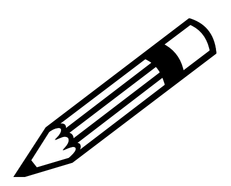
   Ces événements sont souvent implicites dans la phase d'analyse. Pour faciliter l'implantation concrète, il est nécessaire dès maintenant d'indiquer le moment où ces événements auront lieu.

3. Ajouter la direction de navigation aux relations d'association des classes

   Dans le modèle de conception, il est nécessaire d'indiquer explicitement l'orientation des relations. Nous devons indiquer le rôle des classes impliquées dans une relation d'association. Il faut donc ajouter une tête de flèche pour indiquer la cible de la relation. Ce travail facilite l'implantation du modèle de conception lors de la phase de construction où les diagrammes sont traduits en code C++.

4. Ajouter les relations de dépendance dans le diagramme des classes.

   En UML les relations de dépendance (indiquées par des lignes fléchées en pointillées) servent à indiquer qu'un élément UML a la connaissance d'un autre élément UML. Dans le cas des classes, cela signifie qu'une classe est visible par une autre classe. En terme de C++, si A est dépendant de B alors B est visible par A. Autrement dit, la directive `#include <B.h>` est nécessaire dans le fichier source de A.



Les quatre activités énumérées ci-dessus ont pour objectif : *i*) d'identifier les attributs et opérations manquantes; *ii*) d'identifier des classes collection; *iii*) de spécifier les types concrets et le niveau d'accès des attributs et opérations d'après les capacités du langage de programmation; *iv*) de spécifier les contraintes numériques imposées aux attributs (ainsi qu'aux opérations). Parallèlement, il est nécessaire de déterminer les algorithmes associés aux opérations importantes du système. Nous pouvons décrire les algorithmes impliqués à l'aide d'une écriture en pseudo-code ou à l'aide d'ordinogrammes.

5. Choisir sur les éléments de l'interface graphique

   Établir l'apparence et la mécanique de fonctionnement de l'interface utilisateur. Définir les interactions qui existent entre les éléments de l'interface graphique. Dorénavant, les éléments de l'interface graphique font partie intégrante de la conception.

Enfin, toutes les activités énumérées dans cette sous-section peuvent être réalisées en parallèle d'une manière itérative.

9.1.1 MISE À JOUR DES DIAGRAMMES UML

Cette sous-section présente la mise en œuvre pratiques de la réalisation des tâches 1 à 4 de la sous-section 9.1 à l'aide de notre exemple d'application. Il s'agit essentiellement l'ajout des artifices informatiques afin de permettre la résolution de la problématique.

9.1.1.1 EXEMPLE D'APPLICATION

D'abord, la notion du temps est omniprésente dans le système d'ascenseurs. En effet le déplacement des ascenseurs et la durée des portes ouvertes sont une fonction du temps. Ainsi, une horloge doit être instaurée afin de cadencer les actions dépendantes. Pour simplifier l'interprétation des résultats de simulation, nous considérerons les unités de temps normalisées. Par exemple, le déplacement d'un étage prendra 5 unités de temps, l'ouverture des portes durera 3 unités de temps, etc. Donc, les actions importantes du système d'ascenseurs seront pilotées par un signal d'horloge.

En UML, un signal est un type d'événement qui représente un stimulus asynchrone communiqué entre deux objets.

Le signal de l'horloge sera modélisé à l'aide de la notation UML de la manière suivante :

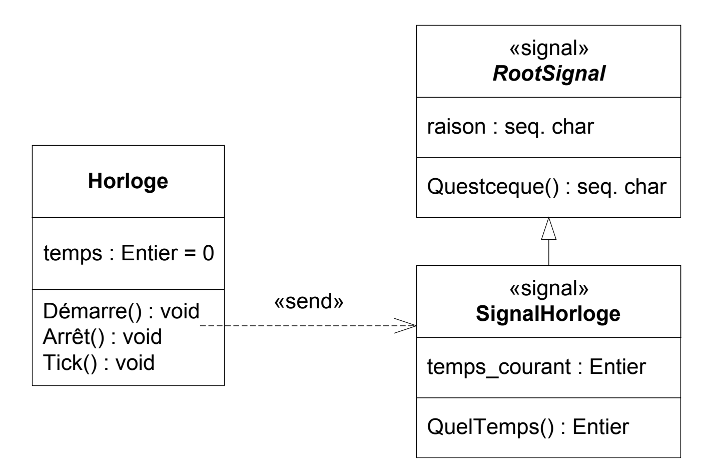


Figure 97 Modélisation de l'horloge et du signal d'horloge.

Ainsi, l'horloge du système est une classe possédant un attribut `temps`. Cet attribut servira à contenir le temps écoulé en unités de temps normalisées (UT). L'opération `Horloge::Tick()` sera enclenchée à toutes les UT provoquant l'envoie du signal `SignalHorloge`. La transmission du signal d'horloge est indiquée par la relation de dépendance marquée par le stéréotype « send ».

En UML, un signal est une classe augmentée par le stéréotype « signal ».

Nous avons instauré une hiérarchie d'un seul niveau pour représenter le signal d'horloge. La classe abstraite `RootSignal` est la classe de base de tous les signaux du système d'ascenseurs. Elle renferme un attribut nommé `raison` qui est une chaîne de caractères ainsi qu'une opération `Questceque()` qui retourne cette chaîne de caractères. Cet attribut sert à expliquer la raison de l'envoie du signal aux objets récepteurs. L'héritage de `RootSignal` produit une seconde classe nommée `SignalHorloge`. Cette classe sera utilisée par `Horloge` pour signaler l'écoulement du temps aux objets du système. Dans signalHorloge, l'attribut `temps_courant` aura la même valeur que `Horloge::temps`. Les objets recevant ce signal peut connaître le temps courant en utilisant l'opération `SignalHorloge::QuelTemps()`. Enfin, la seule distinction entre une classe ordinaire et un signal est que le dernier est décoré par le stéréotype « signal » (voir Figure 97).

Il existe également le mot clé `when(expr)` pour exprimer un moment précis dans le temps.

L'envoi périodique du signal d'horloge par l'objet `Horloge` est illustré par le diagramme d'état de la Figure 98. Le mot clé after permet l'expression de l'écoulement du temps. Ainsi `after(1 UT) / temps += 1` signifie, dans notre contexte, après chaque unité de temps, l'attribut temps est augmenté de une unité de temps. Parallèlement, l'état de l'objet `Horloge` passe de Inactif à Actif. Une fois, rendu dans l'état Actif, l'objet `Horloge` déclenche un signal d'horloge par le biais de l'opération `Tick()`.

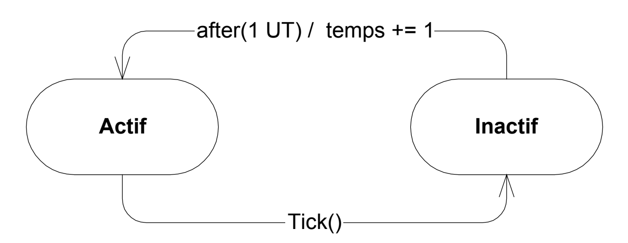


Figure 98 Diagramme d'état de l'horloge.

Dans notre problématique, il est clair que le signal d'horloge sert à cadenser le fonctionnement des ascenseurs. Conséquemment, le récepteur du signal d'horloge est la classe `Ascenseur` ou plus précisément la classe de base `ASC` (voir le diagramme des classes Figure 96). Nous devons donc ajouter la classe `Horloge`, la hiérarchie du signal et les relations appropriées dans le diagramme des classes de l'exemple d'application. À cause de la complexité de ce diagramme, nous ne montrerons que les relations impliquant les classes `Horloge`, `RootSignal`, `SignalHorloge`, `ASC` et `Ascenseur`. Enfin, il est à noter que l'horloge du système d'ascenseurs est gérée par le `Contrôleur`. C'est le Contrôleur qui est responsable du démarrage et de l'arrêt de l'horloge.

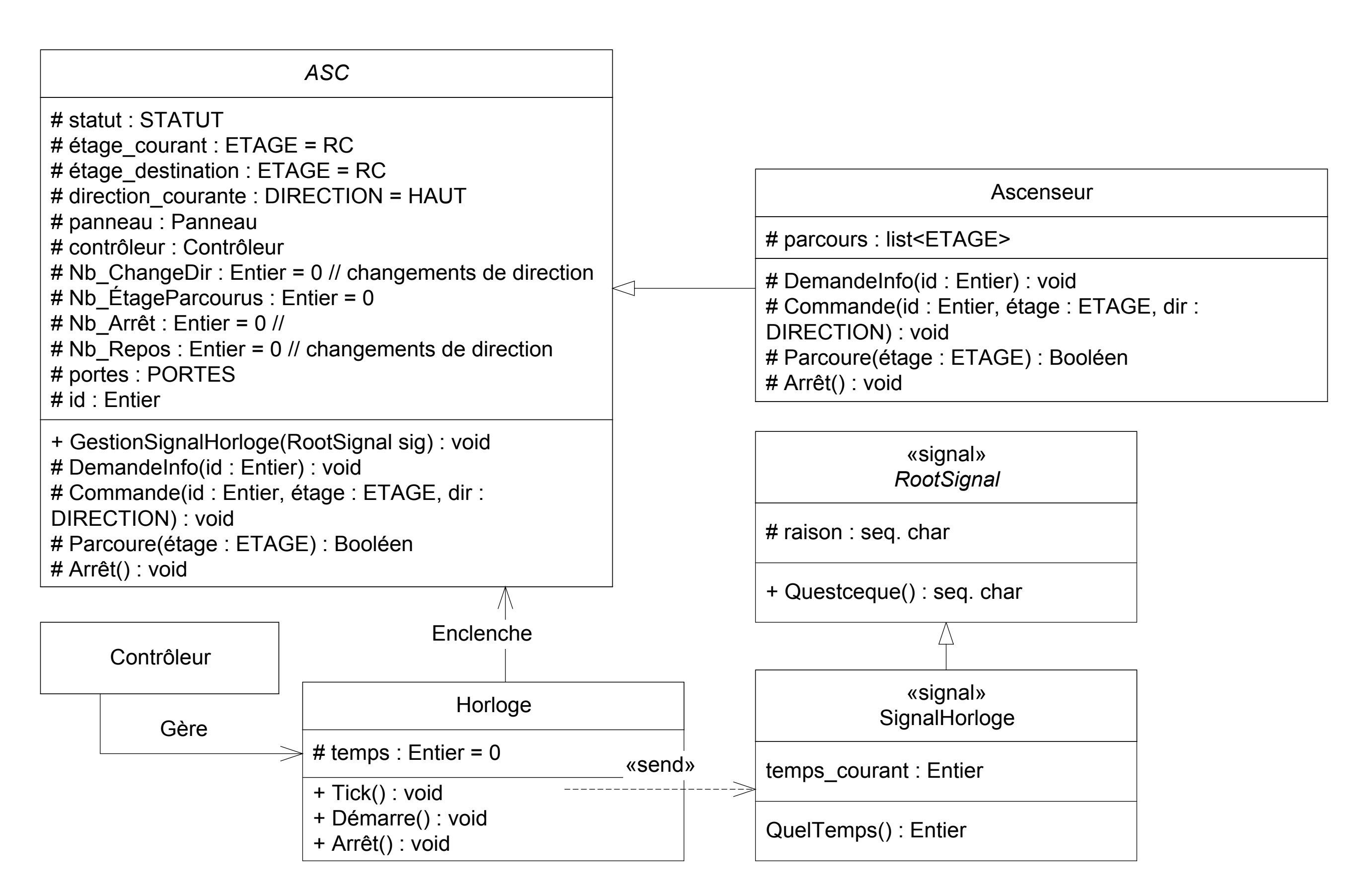


Figure 99 Diagramme de classes illustrant les relations entre le signal d'horloge et la classe de base ASC.

Note de construction : Sous Windows NT/2000/XP, l'horloge est réalisée grâce aux appels de système `SetTimer()` et `KillTimer()`. Le message `WM_TIMER` est alors généré à intervalles réguliers. Nous pouvons ainsi programmer la classe `Horloge` de sorte qu'elle utilise `SetTimer()` pour cadencer l'exécution périodique de sa fonction membre `Tick()`. À son tour, la fonction membre `Tick()` enclenche la fonction membre publique `ASC::GestionSignalHorloge()` en utilisant un objet de type `SignalHorloge` comme paramètre.

Une autre technique de réalisation consiste à invoquer la fonction membre `ASC::GestionSignalHorloge()` indirectement en utilisant un message Windows. Lors de l'exécution de la fonction membre `Horloge::Tick()`, cette dernière envoie un message Windows de type `WM_APP` vers un objet de type `Ascenseur` à l'aide de fonction système `SendMessage()`. Le signal d'horloge est passé en paramètre dans la fonction `SendMessage()` comme un pointeur de 32bits. À la réception du message `WM_APP`, l'objet Ascenseur doit convertir le pointeur de 32bits en un objet de type `SignalHorloge` et exécuter sa fonction membre `GestionSignalHorloge()` en lui passant l'objet `SignalHorloge` comme paramètre.

Enfin, dans les deux techniques mentionnées, la classe `Horloge` doit connaître l'existence de la classe `ASC` d'où la relation d'association joignant ces deux classes.

Voici les diagrammes de collaboration du système en tenant compte de l'instauration d'une horloge.

h1.0: [after(1 UT)] Tick()
c1: Démarre()
h: Horloge
h1.1 GestionSignalHorloge()
a13: Désactiver()
b1: MaJ()
Sélectionner()
b: Bi
: ListeBrn
a9: Parcoure()
a10,
a8: ChangerStatut()
: Statut
a3: ListReq()
Passager
Sélectionner()
f1: MaJ()
a: Ascenseur
f: Fi
a11:
Désactiver()
a12: Désactiver()
b2, f2:
Augmenter_compteur()
a2: DemandeInfo()
a7: Commande()
a1: Attention()
a5: Info()
: Panneau
a6: Décision()
: Règle
c: Contrôleur
a4: FormeListReq()

Figure 100 Diagramme de collaboration du système d'ascenseurs incluant l'objet `Horloge`.

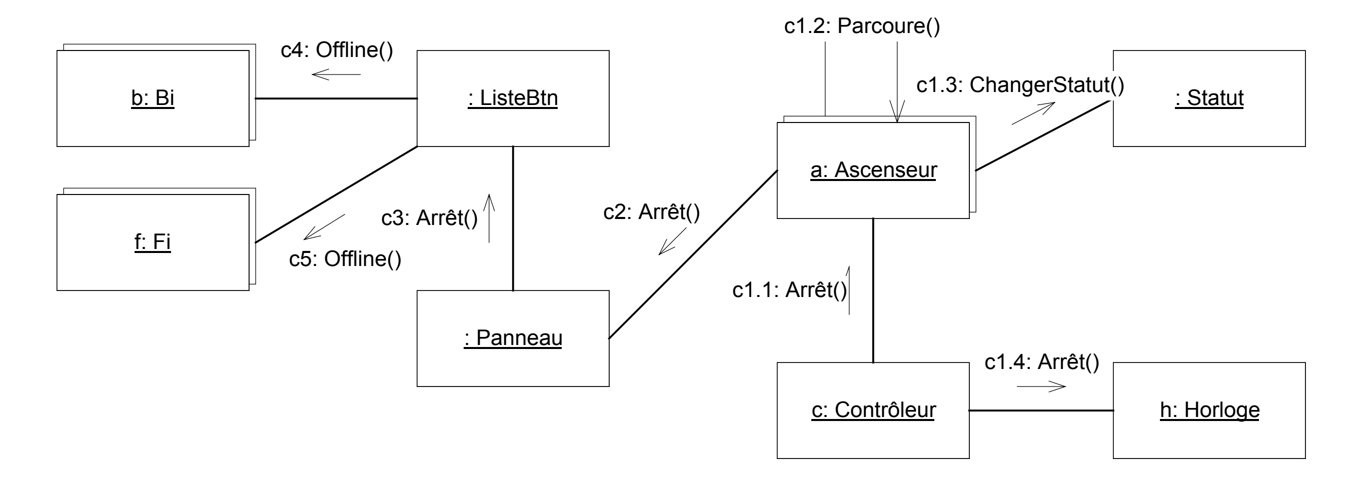


Figure 101 Diagramme de collaboration du système d'ascenseurs montrant l'arrêt de l'horloge par le `Contrôleur` en cas de défaillance.

9.1.2 STRUCTURES DE DONNÉES ET ALGORITHMES

Il est faux de prétendre que l'approche orientée objet élimine l'utilité de la connaissance des structures de données et algorithmes.

Lors de la phase d'analyse, le point important était de montrer ce que le système doit faire en attribuant des responsabilités aux concepts dégagés. Il n'était pas question de dicter le mécanisme exact par lequel les concepts doivent entreprendre ses actions. Or, le « comment faire » est essentiel à la résolution de la problématique.

L'objectif de cette sous-section est de dégager les structures de données et les algorithmes selon une approche orientée objet. Dans un premier temps, il est important de bien connaître les structures et algorithmes classiques existants. Surtout leurs champs d'application et leur complexité spatiale et temporelle.

D'un point de vue orienté objet, les structures de données ne sont pas des entités passives. Prenons l'exemple des classes collection et de séquence de la bibliothèque STL (voir chapitre 2, sous-section 2.2). Elles encapsulent les données et disposent d'un ensemble d'opérations dédidées. Ces opérations facilitent la création d'algorithmes en leur offrant des services efficaces et nécessaires dans la manipulation des données. Ainsi, les algorithmes ne sont pas considérées comme des routines agissant sur des structures passives mais bien des échanges de messages entre objets. Les objets réalisant une structure de données sont manipulés par des objets réalisant des algorithmes. Les actions des algorithmes sont distribuées à travers l'interface d'un ensemble d'objets et les données sont encapsulées derrière des services opaques d'un autre ensemble d'objets. Conséquemment, il faut éviter la création d'une immense structure contenant toutes les données à manipuler. De même, il ne faut pas enrober les algorithmes dans une routine à tout faire. Voici quelques conseils qui faciliteront leur réalisation pratique :

- **Identifier les opérations des classes nécessitant l'implantation des algorithmes**. Puisque l'approche orientée objet distribue les responsabilités entre les objets. Ces responsabilités sont réalisées à l'aide d'opérations. Il est donc naturel de commencer la lecture par les opérations des classes.

- **Choisir les structures de données et les algorithmes connus**. Dans la majorité des cas, les structures et les algorithmes classiques suffisent à la tâche. Autrement dit, ne pas réinventer la roue. Consulter le contenu de la bibliothèque STL ou les références portant sur l'analyse des structures de données et algorithmes (GPA665) pour connaître leur l'applicabilité et leur complexité spatiale et temporelle.

- **Porter une préférence aux réalisations standardisées**. Il faut favoriser l'utilisation de la bibliothèque STL plutôt que celle fabriquée maison. Les réalisations standardisées sont normalement plus robustes et versatiles. Nous devons privilégier l'utilisation des classes de collection et des algorithmes génériques de la bibliothèque STL plutôt que de les coder nous-mêmes.

- **Concevoir les structures et algorithmes en utilisant l'approche orientée objet**. Dans le cas où nos besoins particuliers ne peuvent être comblés par des réalisations standardisées, nous devons alors concevoir nous-mêmes ces objets informatiques. La conception repose encore une fois sur les résultats de l'analyse

orientée objet en parcourant les mêmes étapes présentées dans le chapitre 8. Les concepts impliqués sont alors des entités informatiques et les cas d'utilisation sont centrés autour de la description des activités à accomplir par les algorithmes. Éventuellement, nous pouvons déceler les activités de base sur lesquelles reposent nos algorithmes. Ces activités de base (i.e. triage, fouille, transformation, opérations arithmétiques, etc.) sont alors puissées des réalisations standardisées. La conception orientée objet consiste à intégrer les classes des bibliothèques standardisées dans les classes des algorithmes.

9.1.2.1 EXEMPLE D'APPLICATION

Deux opérations du système d'ascenseurs méritent une étude plus approfondie. Elles sont :

- `Ascenseur::Parcoure(ETAGE e) : Booléen`. Son rôle consiste à parcourir les étages à partir de `Ascenseur::étage_courant` jusqu'à l'étage de destination `e`. Cette opération doit comptabiliser le nombre d'étages parcourus par l'ascenseur. Elle doit parcourir les étages selon la liste des étapes desservis `Ascenseur::parcours`. Une fois arrivée à l'étage de destination, elle doit régler son attribut `étage_courant` à la valeur de l'étage de destination. Enfin, le déplacement de l'ascenseur est cadencé par un signal d'horloge.

Pseudo-code de la fonction membre `Parcoure()` de la classe `Ascenseur`.

```
Ascenseur::Parcoure(ETAGE e) : Booléen, cadensé par GestionSignalHorloge()
{
 // Rendu à la destination ?
 if (e == étage_courant) {
  statut = ARRÊT;
  Parcoure_terminé = VRAI;
  return Parcoure_terminé;
 }
 Parcoure_terminé = FAUX
 statut = EN_MARCHE
 // Déplacer l'ascenseur à l'étage suivant ?
 if (temporisation_déplacement != DEPLACEMENT_UT) {
  temporisation_déplacement ++;
  return Parcoure_terminé;
 }
 // Trouver le prochain étage dans la liste des étages desservis
 // selon la direction de parcours
 // parcours est une liste STL
 iterator i = find(parcours.begin(), parcours.end(), étage_courant);
 if (direction_courante == HAUT)
  étage_courant = parcours(i + 1);
 else
  étage_courant = parcours(i - 1);
 Nb_ÉtageParcourus++;
 temporisation_déplacement = 0;
 return Parcoure_terminé;
}
```

Dans cet algorithme, nous avons créé deux nouvelles variables : `Parcoure_terminé : Booléen` et `temporisation_déplacement : Entier`. La variable `Parcoure_terminé` indique la fin d'un déplacement (à partir de l'étage courant jusqu'à l'étage de destination). La variable `temporisation_déplacement` est un compteur qui accumule les unités de temps afin de simuler le temps nécessaire

pour le déplacement d'un étage. Ainsi, l'ascenseur doit attendre DEPLACEMENT_UT unités de temps afin de progresser à l'étage suivant. La valeur de DEPLACEMENT_UT doit être réglée proprement dans le codage. Enfin, nous avons appliqué l'algorithme générique `find()` sur l'attribut `parcours` pour trouver l'étage suivant. Évidemment, l'application de `find()` suppose que l'attribut `parcours` est un objet collection de la bibliothèque STL.

- `Ascenseur::GestionSignalHorloge(RootSignal sig) : void`. Son rôle consiste à coordonner les activités temporelles de l'ascenseur. Ces activités temporelles sont :

  1. Déplacement de l'ascenseur. Un nombre d'unités de temps sont nécessaires pour parcourir un étage.

  2. Ouverture des portes. Lorsque l'ascenseur est à l'arrêt. Ses portes doivent demeurer en position ouverte pendant une certaine durée de temps.

Pseudo-code de la fonction membre `GestionSignalHorloge()` de la classe `Ascenseur`.

```
Ascenseur::GestionSignalHorloge() : void, cadensé par le signal d'horloge
{
 // Gérer les portes de l'ascenseur ?
 if (statut == ARRÊT) {
  // les portes sont-elles ouvertes ?
  if (portes == OUVERTES) {
   // faut-il les fermer maintenant ?
   if (temporisation_portes == PORTES_UT) {
    temporisation_portes = 0;
    portes = FERMÉES;
    statut = EN_MARCHE;
    return;
   } else {          // maintenir les portes ouvertes
    temporisation_portes++;
    return;
   }
  } else { // les portes étaient-elles fermées ?
   portes = OUVERTES;
   return;
  }
  // fin du traitement
  return;
 } else { // l'ascenseur en déplacement
  Ascenseur::Parcoure(étage_destination);
  return;
 }
}
```

Dans cet algorithme, nous avons créé une nouvelle variable : `temporisation_portes : Entier`. La variable `temporisation_portes` est un compteur qui accumule les unités de temps afin de simuler le temps nécessaire pour maintenir les portes de l'ascenseur ouvertes. Ainsi, les portes demeurent ouvertes pendant PORTES_UT unités de temps afin de permettre l'embarquement et le débarquement des passagers. La valeur de PORTES_UT doit être réglée proprement dans le codage.

9.1.3 IDENTIFICATION DES ATTRIBUTS ET OPÉRATIONS MANQUANTS

Cette tâche est nécessaire pour rendre complet le modèle de conception. En effet, certains attributs et opérations ne sont pas indiqués dans le modèle d'analyse afin de construire un modèle conceptuel qui est intelligible. Ces détails délaissés lors de la l'analyse doivent être complétés dans la phase de la conception.

De même, certains attributs et opérations sont créés lors de l'élaboration des algorithmes de solution. Puisque les algorithmes utilisés ne sont pas anticipés lors de la phase d'analyse, il s'avère nécessaire de les ajouter dans la phase de conception. Peu importe les raisons qui motivent leur création, les nouveaux attributs et opérations doivent toujours contribuer à la solution de la problématique.

Une technique pratique et efficace consiste à passer en revue les attributs dégagés lors de l'analyse et d'y associer les opérations d'accès (set/get) appropriées. Ainsi, un attribut est exposé à un autre objet s'il est utilisé par un autre objet. Dans ce cas, une opération d'accès doit être créée pour permettre l'utilisation de l'attribut. Cette opération d'accès doit être publique tandis que l'attribut lui-même doit être protégé ou privé selon le cas.

Pour les attributs, ils sont ajoutés en fonction des algorithmes à réaliser. Ces nouveaux attributs servent donc aux algorithmes et sont ajoutés dans la section protégée ou privée selon le cas.

Un conseil : Ne pas laisser les attributs dans la section publique des classes. Toujours assigner les attributs dans la section protégée ou privée selon le cas. Ajouter des opérations d'accès au besoin. Enfin, l'identification des attributs et des opérations est un processus itératif. Il est un devoir de mettre à jour la documentation afin de refléter le modèle de conception.

Résumons cette technique succinctement :

- Ajout des opérations : Inspecter les attributs des classes. Instaurer des opérations d'accès dans la section publique des classes pour les attributs qui sont utilisés par d'autres classes.
- Ajout des attributs : Créer les algorithmes nécessaires. Instaurer des opérations d'accès dans la section publique ou protégée afin de permettre l'accès des attributs servant aux algorithmes.
- Effectuer ces ajouts en parallèle d'une manière itérative.

9.1.3.1 EXEMPLE D'APPLICATION

Nous reprenons l'exemple d'application présenté dans les chapitres précédents. Les attributs de base ont été dégagés lors de la phase d'analyse (voir Tableau 22). Nous

devons étudier attentivement l'utilisation de ces attributs dans le contexte des diagrammes d'interaction obtenus lors de l'analyse orientée objet (voir Figure 88 à Figure 90). Le but est d'identifier les opérations d'accès aux attributs des classes.

Attribut	**Opérations d'accès**
Bouton::état	Get_état() : ETAT Set_état(ETAT état) : void
Bouron::étage	Get_étage() : ETAGE Set_étage(ETAGE état) : void
Bouton::direction	Get_direction() : DIRECTION Set_direction(DIRECTION dir) : void
Bouton::lumière1	Get_lumière1() : Booléen Set_lumière1(Booléen v) : void
Fi::lumière2	Get_lumière2() : Booléen Set_lumière2(Booléen v) : void
ListeBtn::boutonBi	Get_boutonBi() : list<Bouton>
ListeBtn::boutonFi	Get_boutonFi() : list<Bouton>
ListeBtn::Nb_BoutonActif	Get_Nb_BoutonActif() : Entier Set_ Nb_BoutonActif(Entier v) : void
ListeBtn::état	Get_état() : ETAT Set_état(ETAT état) : void
Note : les attributs boutonBi et boutonFi seront assignés lors de la création de l'objet ListeBtn.	
Panneau::boutons	Get_ListBtn() : ListeBtn
Panneau::état	Get_état() : ETAT Set_état(ETAT état) : void
Panneau::Nb_Selection	Get_Nb_Selection() : Entier Set_ Nb_Selection(Entier v := 0) : void
Note : l'attribut boutons sera assigné lors de la création de l'objet Panneau.	
ASC::statut	Get_statut() : STATUT Set_statut(STATUT stat := REPOS) : void
ASC::étage_courant	Get_étage_courant() : ETAGE Set_étage_courant(ETAGE e := RC) : void
ASC::étage_destination	Get_étage_destination() : ETAGE Set_étage_destination(ETAGE e := RC) : void
ASC::direction_courante	Get_direction_courante() : DIRECTION Set_direction_courante(DIRECTION d = HAUT) : void
ASC::panneau	Get_panneau() : Panneau
ASC::contrôleur	Get_contrôleur() : Contrôleur
ASC::Nb_ChangeDir	Get_Nb_ChangeDir() : Entier Set_Nb_ChangeDir(Entier v := 0) void
ASC::Nb_ÉtageParcourus	Get_Nb_ÉtageParcourus() : Entier Set_Nb_ÉtageParcourus(Entier v := 0) void
ASC::Nb_Arrêt	Get_Nb_Arrêt() : Entier Set_Nb_Arrêt(Entier v := 0) void
ASC::Nb_Repos	Get_Nb_Repos() : Entier Set_Nb_Repos(Entier v := 0) void

ASC::portes	Get_portes() : PORTES Set_portes(PORTES p = FERMÉES) : void
ASC::id	Get_id() : Entier Set_id(Entier v) : void
ASC::parcoure_terminé	Attribut interne
ASC::temporisation_déplacement	Attribut interne
ASC::temporisation_portes	Attribut interne
Note : Les attributs panneau et contrôleur seront assignés lors de la création de l'objet ASC par le biais de l'objet Ascenseur.	
Ascenseur::parcours	Get_parcours() : list<ETAGE> Set_parcours(list<ETAGE> l) : void
Note : L'attribut parcours peut être assigné lors de la création de l'objet Ascenseur.	
Contrôleur:: ascenseurs	Get_ascenseurs() : vector<ASC> Set_ascenseurs(vector<ASC> v) : void Get_ascenseurs(Entier i) : ASC Set_ascenseurs(Entier i, ASC a) : void
Contrôleur:: défaillance	Get_défaillance() : Booléen Set_défaillance(Booléen := FAUX) : void
Contrôleur:: règle	Get_règle() : Règle
Contrôleur:: état_traitement	Get_état_traitement() : vector<TRAITEMENT> Set_ état_traitement(vector<TRAITEMENT>) : void Get_état_traitement(Entier i) : TRAITEMENT Set_ état_traitement(Entier i, TRAITEMENT t) : void
Note : L'attribut régle sera assigné lors de la création de l'objet Contrôleur.	
Règle::numéro_asc	Get_numéro_asc() : Entier Set_numéro_asc(Entier v) : void
Règle::requêtes	Get_requêtes() : list<ETAGE, DIRECTION> Set_requêtes(list<ETAGE, DIRECTION> l) : void
Règle::direction_courante	Get_direction_courante() : DIRECTION Set_direction_courante(DIRECTION dir) : void
Règle::prochain_etage	Get_prochain_etage() : ETAGE Set_prochain_etage(ETAGE e) : void
Statut::statut_courant	Get_statut_courant() : STATUT Set_statut_courant(STATUT stat) : void
Statut::statut_précédent	Get_statut_précédent() : STATUT Set_statut_précédent(STATUT stat) : void
DONNÉES::étage_courant	Get_étage_courant() : ETAGE Set_étage_courant(ETAGE e) : void

DONNÉES::direction_courante DONNÉES::liste_sel	Get_direction_courante() : DIRECTION Set_direction_courante(DIRECTION dir) : void Get_liste_sel() : list<ETAGE, DIRECTION> Set_liste_sel(list<ETAGE, DIRECTION> l) : void
DÉCISION:: étage_destination DÉCISION::direction	Get_étage_destination() : ETAGE Set_étage_destination(ETAGE e) : void Get_direction() : DIRECTION Set_direction(DIRECTION dir) : void

Tableau 46 Opérations d'accès des attributs de l'exemple d'application à l'état actuel de la conception.

9.1.4 SPÉCIFICATION DES CONTRAINTES

Parmi les activités de la phase de conception la spécification des contraintes est sans doute celle qui est la plus négligée. Curieusement, une bonne spécification des contraintes peut éliminer une grande partie des problèmes reliés à la construction du logiciel à partir du modèle de conception.

Pour les attributs des classes, la spécification des contraintes concerne les valeurs limites, la quantité et la durée temporelle de ces attributs. On indiquera les contraintes sur les attributs par des tableaux ou directement dans les diagrammes UML. Pour les opérations, la spécification des contraintes concerne la pré-condition et la post-condition de ces opérations. Les contraintes sur les opérations sont normalement indiquées dans le contract des opérations (voir sous-section 8.5.1).

9.1.4.1 EXEMPLE D'APPLICATION

Dans cette sous-section nous présenterons les contraintes des attributs de notre exemple d'application (système d'ascenseurs).

Attribut	Contraintes
Bouton::état Bouron::étage Bouton::direction Bouton::lumière1	état ∊ {activé, désactivé, hors_ligne} étage ∊ {S2, S1, RC, E1, M1, E2, M2, E3, M3} Pour boutons Bi direction == HAUT_BAS, pour boutons Fi direction ∊ {HAUT, BAS} lumière1 ∊ {VRAI, FAUX}
Fi::lumière2	lumière2 ∊ {VRAI, FAUX}
ListeBtn::boutonBi	Pour ascenseur A1, boutonBi contient les boutons représentant les étages S2, S1, RC, E1, M1, E2, M2, E3 et M3. Pour l'ascenseur A2, boutonBi contient les boutons représentant les étages S1, RC, E1, E2 et E3. Pour l'ascenseur A3, boutonBi contient les boutons représentant les étages RC, E1, E2 et E3.

ListeBtn::boutonFi	Pour ascenseur A1, boutonFi contient les boutons représentant les étages S2, S1, RC, E1, M1, E2, M2, E3 et M3. Pour l'ascenseur A2, boutonFi contient les boutons représentant les étages S1, RC, E1, E2 et E3. Pour l'ascenseur A3, boutonFi contient les boutons représentant les étages RC, E1, E2 et E3.
ListeBtn::Nb_BoutonActif	0 ≤ Nb_BoutonActif ≤ MAX_UINT
ListeBtn::état	état ∊ {activé, désactivé, hors_ligne}
Panneau::boutons	Un objet de type ListeBtn.
Panneau::état	état ∊ {activé, désactivé, hors_ligne}
Panneau::Nb_Selection	0 ≤ Nb_Selection ≤ MAX_UINT
ASC::statut	statut ∊ {REPOS, ARRÊT, EN_MARCHE, DÉFAILLANCE}
ASC::étage_courant	étage_courant ∊ Ascenseur::parcours.
ASC::étage_destination	étage_destination ∊ Ascenseur::parcours.
ASC::direction_courante	direction_courante ∊ {HAUT, BAS}
ASC::panneau	Un objet de type Panneau.
ASC::contrôleur	Un objet de type Contrôleur.
ASC::Nb_ChangeDir	0 ≤ Nb_ChangeDir ≤ MAX_UINT
ASC::Nb_ÉtageParcourus	0 ≤ Nb_ÉtageParcourus ≤ MAX_UINT
ASC::Nb_Arrêt	0 ≤ Nb_Arrêt ≤ MAX_UINT
ASC::Nb_Repos	0 ≤ Nb_Repos ≤ MAX_UINT
ASC::portes	portes ∊ {OUVERTES, FERMÉES}
ASC::id	Pour cette version du logiciel, 1 ≤ id ≤ 3
ASC::parcoure_terminé	parcoure_terminé ∊ {VRAI, FAUX}
ASC::temporisation_déplacement	0 ≤ temporisation_déplacement ≤ MAX_UINT
ASC::temporisation_portes	0 ≤ temporisation_portes ≤ MAX_UINT
Ascenseur::parcours	parcours est une liste d'étages. Voir BoutonBi et BoutonFi.
Contrôleur::ascenseurs	Pour cette version du logiciel, il y a 3 ascenseurs dans le système et ascenseurs est un vecteur de 3 éléments de type ASC.
Contrôleur::défaillance	défaillance ∊ {VRAI, FAUX}
Contrôleur::règle	Un objet de type Règle.
Contrôleur::état_traitement	Pour cette version du logiciel, était_traitement est un vecteur de 3 éléments de type TRAITEMENT où TRAITEMENT ∊ {SOLLICITATION_REÇUE, DEMANDE_INFO, CONSULTE_RÈGLE, COMMANDE, REQUÊTE_TRAITÉE}
Règle::numéro_asc	Pour cette version du logicielle, 1 ≤ numéro_asc ≤ 3.
Règle::requêtes	requêtes est une liste contenant un nombre indéterminé d'éléments. Chaque élément est un pair (ETAGE, DIRECTION).
Règle::direction_courante	direction_courante ∊ {HAUT, BAS}
Règle::prochain_etage	prochain_étage ∊ {S2, S1, RC, E1, M1, E2, M2, E3, M3}
Statut::statut_courant	statut_courant ∊ {REPOS, ARRÊT, EN_MARCHE, DÉFAILLANCE}

Statut::statut_précédent	statut_précédent ∊ {REPOS, ARRÊT, EN_MARCHE, DÉFAILLANCE}
DONNÉES::étage_courant DONNÉES::direction_courante DONNÉES::liste_sel	étage_courant ∊ {S2, S1, RC, E1, M1, E2, M2, E3, M3} direction_courante ∊ {HAUT, BAS} liste_sel est une liste contenant un nombre indéterminé d'éléments. Chaque élément est un pair (ETAGE, DIRECTION).
DÉCISION:: étage_destination DÉCISION::direction	étage_destination ∊ Ascenseur::parcours. Autrement dit, l'étage de destination doit être un étage desservi par l'ascenseur en question. direction ∊ {HAUT, BAS}
MAX_UINT est la valeur maximale de unsigned int en C++.	

Tableau 47 Contraintes sur les attributs de l'exemple d'application.

9.1.5 DESTRUCTION ET CRÉATION DES OBJETS

Le moment où un objet est créé (ou détruit) est largement déterminé par les relations qui existent et les patrons de conception utilisés (voir sous-section 9.2). D'une façon générale, un objet de type ObjA est responsable de la création d'un objet de type ObjB si l'une des conditions suivantes est vraie :

- ObjA est en agrégation avec ObjB;
- ObjA contient ObjB;
- ObjA utilise ObjB;
- ObjA possède les paramètres d'initialisation de ObjB.

Consulter le chapitre 6 pour connaître les détails techniques reliés à la création (destruction) des objets impliqués dans une relation orientée objet.

L'application de patrons de conception dans la conception peut modifier l'ordre de création (et destruction) des objets. La raison est que ces patrons introduits de nouvelles classes et de nouvelles relations dans le modèle de conception. Consulter la sous-section 9.2 pour de plus ample détails sur la mécanique et l'application des patrons de coneption.

9.1.5.1 EXEMPLE D'APPLICATION

Voici un tableau montrant la responsabilité et le moment de création (et destruction) des objets de notre exemple d'application. Les résultats de ce tableau ne tiennent pas compte de l'application des patrons de conception.

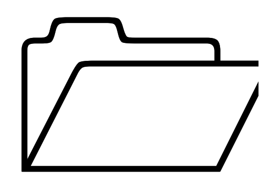
Instance de type	Créer par (quand)	Détruit par (quand)
Ascenseur	Programmer (Début)	Programme (Fin)
Bi, Fi	ListeBtn (Initialisation)	ListeBtn (Destruction)

Contrôleur	Programme (Début)	Programme (Fin)
DÉCISION	Règle (Décision())	Contrôleur (Décision())
DONNÉES	Ascenseur (Info())	Règle (Décision())
Horloge	Contrôleur (Initialisation)	Contrôleur (Destruction)
ListeBtn	Panneau (Initialisation)	Panneau (Destruction)
Panneau	Ascenseur (Initialisation)	Ascenseur (Destruction)
Règle	Programme (Début)	Programme (Fin)
SignalHorloge	Horloge (Tick())	Ascenseur (GestionSignalHorloge())
Statut	Ascenseur (Initialisation)	Ascenseur (Destruction)

Tableau 48 Création et destruction des objets de l'exemple d'application (excluant les patrons de conception).

9.1.6 CONCEPTION DE L'INTERFACE GRAPHIQUE

L'ergonomie des interfaces graphiques est un sujet d'étude pluridisciplinaire alliant la psychologie industrielle, l'infographie, pédagogie et la programmation. Dans cette sous-section, nous ne ferons que souligner les points les plus intuitifs de cette discipline fort utile.



La conception de l'interface graphique débute par l'établissement d'un modèle d'utilisation du logiciel. Le modèle d'utilisation doit viser la simplicité en réduisant au minimum le nombre de paramères et le nombre de « clics » nécessaires pour enclencher une action du logiciel. Nous pouvons construire le modèle d'utilisation en répondant aux questions suivantes :



1. Comment présenter à l'utilisateur les actions disponibles ?

   De nos jours, la réponse à cette question est évidente. La présentation des actions disponibles sera réalisée par des éléments de l'interface graphique (X/Motif, Windows, etc.).

2. Comment indiquer les étapes logiques pour accomplir une tâche ?

   Les étapes doivent être intuitives. Dans la mesure du possible, l'utilisateur doit pouvoir accomplir les tâches sans une lecture exhaustive du manuel d'opération. Pour faciliter l'accomplissement de cet objectif, nous pouvons regrouper les paramètres nécessaires pour une tâche dans un même panneau de paramètres. Mieux encore, créer des Wizards pour aider les utilisateurs à accomplir des tâches spécifiques.

3. Comment l'utilisateur peut-il enclencher une action ?

   La façon dont une action est enclenchée doit être appliquée partout dans le logiciel. Autrement dit, un clic sur un bouton, la touche ENTER ou d'autres manipulations permettant l'enclenchement d'une action. Appliquer le schème systématiquement et minimiser les exceptions (c'est-à-dire, un clic dans un cas

mais il faut un double-clic dans tel autre cas ne fait que créer de la confusion chez l'utilisateur).

4. Comment indiquer la progression d'une action enclenchée ?

   Une contre-réaction visuelle est nécessaire pour toute action du logiciel qui possède une certaine durée. Encore une fois la contre-réaction visuelle doit être appliquée de façon systématique.

5. Comment l'utilisateur peut-il arrêter prématurément l'action déjà enclenchée ou sur le point d'être enclenchée ?

   Des échappatoires doivent être prévues pour permettre l'annulation d'une tâche. Par exemple, disposer d'un bouton libellé « Annuler » dans tous les panneaux de paramètres. Rendre la touche ESC équivalente à la sélection du bouton « Annuler ».

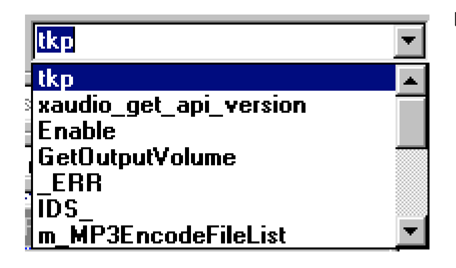
6. Comment signaler la fin d'une action à l'utilisateur ?

   Il faut bien réfléchir sur cette dernière question. En effet, elle renferme deux sous questions. Doit-on toujours avertir l'utilisateur de l'accomplissement d'une tâche ? Quel moyen utilise-t-on pour signaler cet avertissement ? Le signalement de la fin d'une tâche peut vite devenir un ennui pour les utilisateurs. Dans ce cas, il est préférable de permettre à l'utilisateur d'activer ou désactive le signal par une option de configuration du logiciel. Le signalement sonore est à proscrire excepté dans des applications particulières où l'utilisateur n'a pas l'accès direct de l'écran[1].

9.1.6.1 QUELQUES RECOMMANDATIONS



- Le contrôle Bouton doit toujours enclencher une action visible. Sans quoi l'utilisateur peut créer une image mentale incorrecte du système.



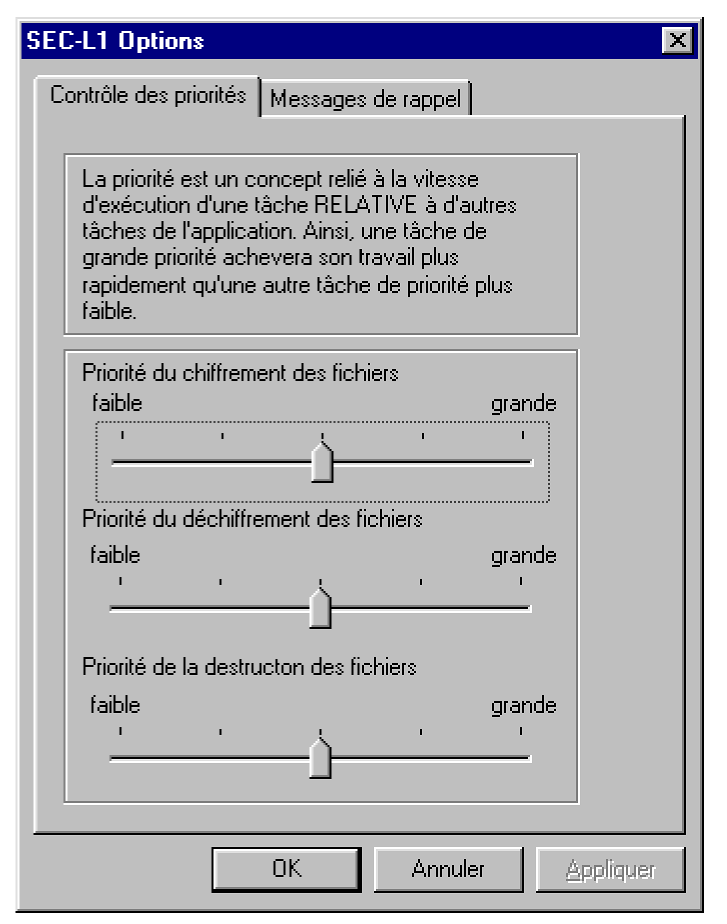
- Utiliser une liste déroulante avec zone d'édition si cette dernière peut réellement faciliter la tâche. Si non, rendre cette zone non éditable.



[1] Ne vous fiez pas au signal sonore accompagnant l'affichage des pannaux de paramètres. L'utilisateur peut facilement désactiver ce signal sonore grâce à la configuration de l'interface graphique du système informatique.

- Le contrôle barre d'outils peut être statique ou flottant. Dans ce dernier cas, prévoir une façon de la faire réapparaître après sa fermeture.

- Mettre le minimum de texte dans un panneau de message et éviter l'utilisation de double négation dans vos textes. Ne pas réinventer la roue, utiliser les panneaux « préfabriqués » de la plateforme. Ceci diminuera le temps d'apprentissage du logiciel. Enfin, éviter les décorations inutiles qui peuvent agacer les utilisateurs. Regrouper les paramètres de configuration du logiciel dans un panneau de paramètres avec onglets. La configuration du logiciel est alors centralisée en un seul endroit. Cela facilitera l'utilisation du logiciel par les utilisateurs.



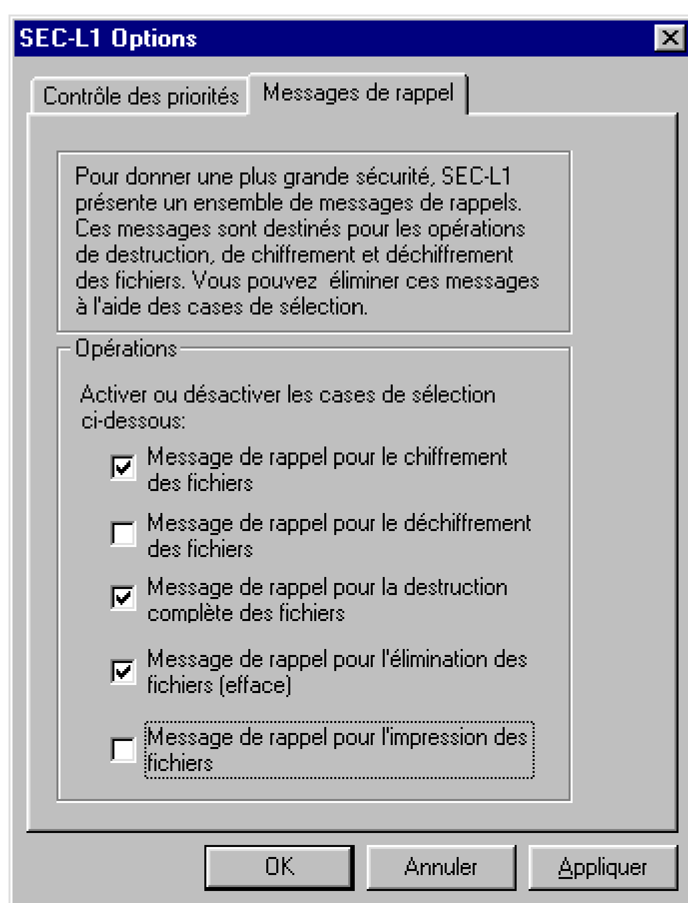


Figure 102 Regrouper les paramètres de réglagle dans un panneau avec onglets.

- Présenter des vues différentes dans une même fenêtre peut réduire la complexité d'utilisation du logiciel. La figure illustre une application comportant trois vues différentes mais physiquement liées. La logique de cette application est rendue évidente par le rôle joué par ses vues. Ainsi, on peut sélectionner un disque ou un dossier dans la vue de gauche. On peut sélectionner un fichier pour traitement ultérieur dans la vue du haut. Enfin, on peut composer, lire et enregistrer un texte dans la vue du bas.

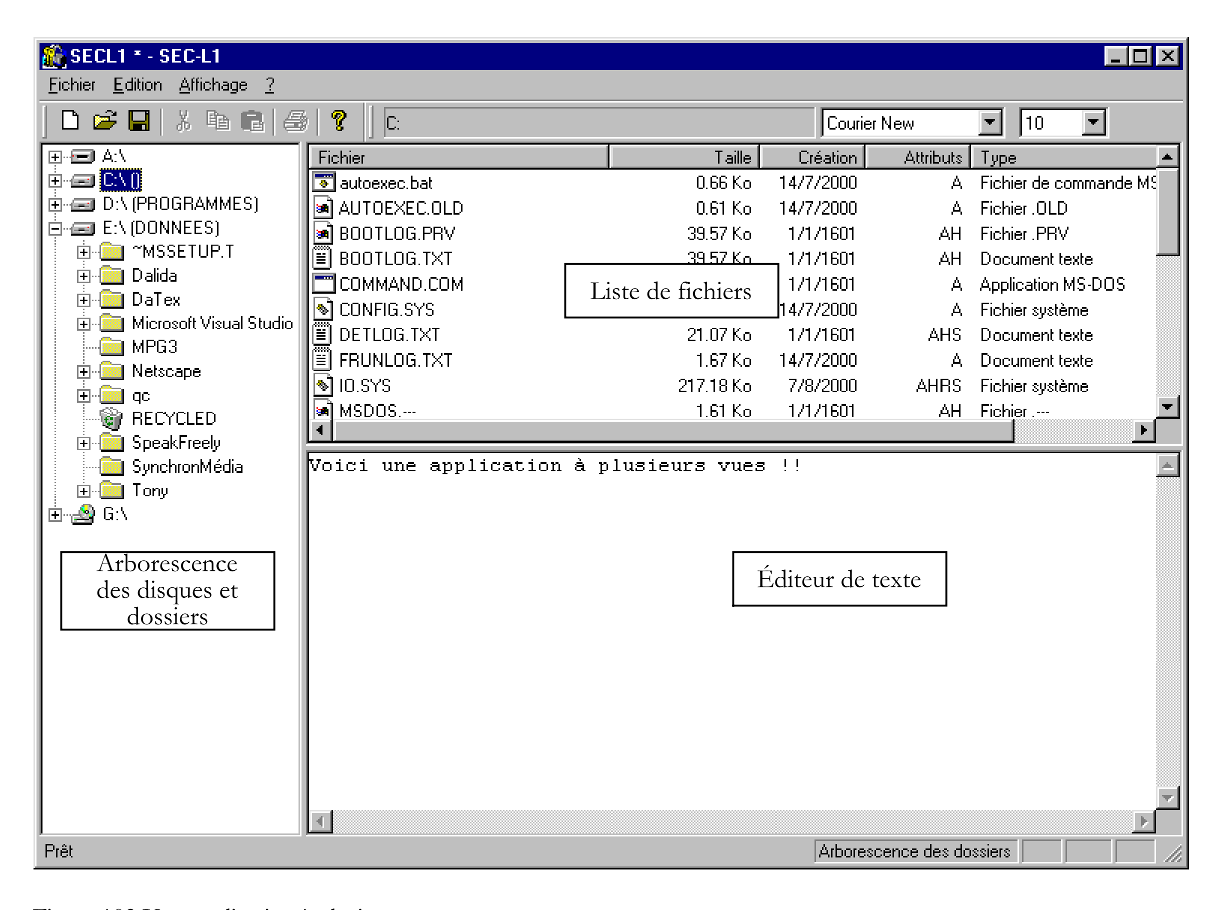
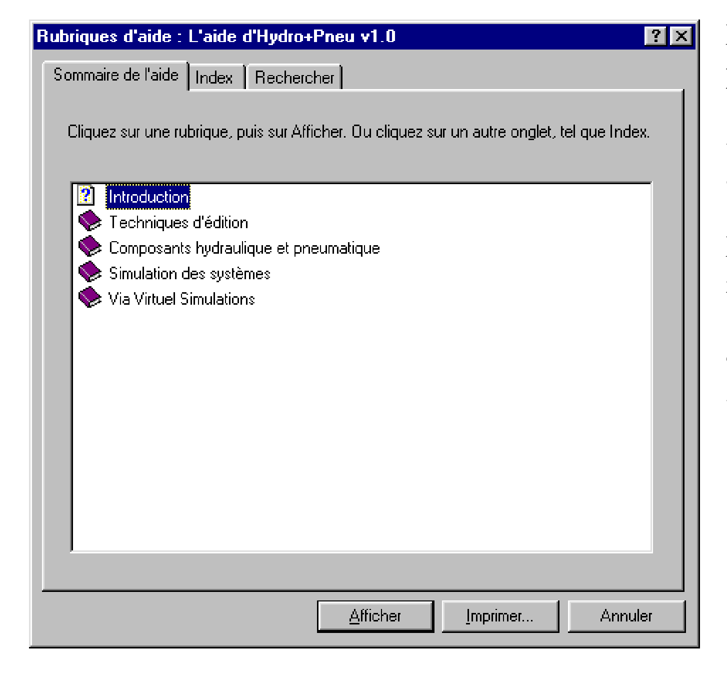


Figure 103 Une application à plusieurs vues.



L'aide en-ligne est primordiale. Un logiciel sans l'aide en-ligne est analogue à une ville sans carte routière.

L'aide statique est de base mais ce qui est mieux est sans doute l'aide dynamique (*context-sensitive help*).

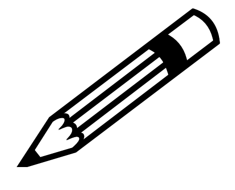
9.2 PATRONS DE CONCEPTION

Les patrons de conception ne constituent pas un concept nouveau. La plupart du temps nous les utilisons sans vraiment prendre le temps de les reconnaître. Cette application inconsciente des patrons de conception peut donner un résultat robuste et efficace mais malheureusement non reproductible. Autrement dit, on ne peut garantir la robustesse et l'efficacité de nos applications logicielles puisque l'on ne sait pas exactement les causes de ce résultat recherché. Parfois, nous avons l'intuition où une certaine construction donne de bons résultats dans une situation donnée. Cette intuition est informelle et personnelle. Elle est difficilement transmissible et sa communication aux membres de l'équipe de développement demeure subjective et ad hoc. L'étude des patrons de conception est donc une façon de formaliser ces constructions « qui marchent » afin de les rendre accessibles à tous les praticiens du domaine.

Encore une fois, les patrons de conception ne sont pas nouveaux. Ils sont obtenus en analysant la structure des logiciels et décelant les constructions qui les rendent robustes et efficaces. Le produit résultant de cette analyse est un ensemble de patrons de conception qui sont :

- **Des solutions réutilisables**. Des patrons généraux applicables dans des contextes bien définis indépendants de la nature des applications et du langage de programmation. De plus, ces patrons sont éprouvés et ont subi avec succès des tests réels dans la pratique. Donc, leur robustesse est garantie.

- **Des terminologies communes**. Chaque patron porte un nom accepté et possède un champ d'application bien cerné. Donc, la communication est facile puisque les praticiens peuvent désormais adopter un vocabulaire et un point de vue commun dans la création de leurs conceptions.



De plus, l'analyse de ces patrons a identifié clairement trois stratégies qui peuvent aider à la conception orientée. Ces stratégies sont :

1. Concevoir pour être interfacé.

   L'approche orientée objet préconise la collaboration des objets dans l'accomplissement des tâches. La conception doit donc davantage axée sur la collaboration entre objets et non sur des objets autonomes. Il est donc recommandé de toujours concevoir les classes en imaginant la mécanique de collaboration liant ces classes.

2. Privilégier la composition (l'agrégation) plutôt que l'héritage.

   L'idée est de restreindre la *profondeur* de la hiérarchie d'héritage en maintenant le nombre de niveaux à un minimum. Une hiérarchie peu profonde facilitera la maintenance et l'expansion future du logiciel. Évidemment, on ne peut

remplacer l'héritage par la composition (l'agrégation); ils ont des sémantiques différentes. Par contre, dans bien des situations, il est possible de choisir entre l'héritage et la composition. Dans de telles situations, privilégier la composition (l'agrégation) au détriment de l'héritage.

3. Identifier les variations puis les encapsuler.

   Pour pouvoir identifier « ce qui varie », nous devons d'abord cerner « ce qui est commun ». Donc, le premier pas à franchir est de recenser les *choses* qui sont de mêmes familles dans le domaine du problème ou dans le domaine de la solution. À partir de ces choses bien déterminées, il est facile d'entrevoir les différences ou les variations qui existent entre les membres de la même famille. Nous pouvons alors créer une encapsulation par famille pour contenir ses variations.

   Ici, l'encapsulation est prise au sens large du terme. Un objet client A accédant un ensemble d'objets à travers une interface opaque est considéré une sorte d'encapsulation. Le point important est que l'objet client A ne doit pas avoir de connaissance directe des objets qu'il a affaire. Les variations quant à elles sont réalisées par la composition (l'agrégation) ou par l'héritage. Évidemment, nous favorisons la composition (l'agrégation) dans la mesure du possible. La Figure 104 illustre la stratégie de l'encapsulation des variations.

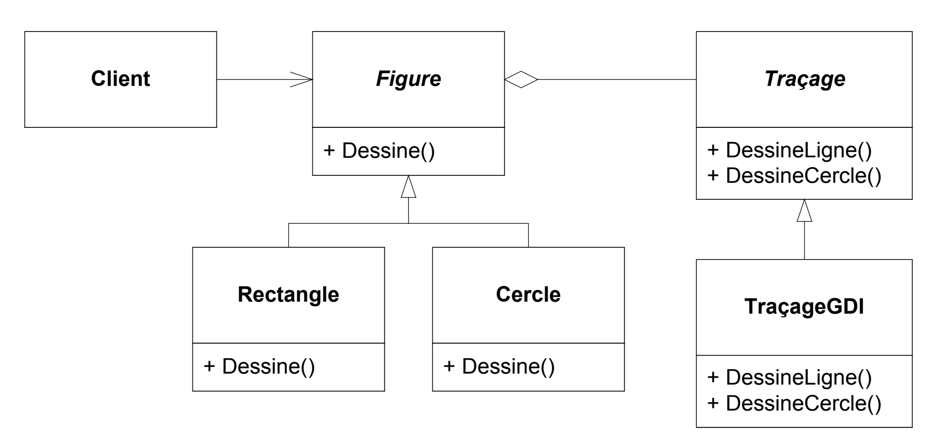


Figure 104 Encapsulation des variations.

   Le client peut accéder aux figures géométriques `Rectangle` et `Cercle`. Cependant, il ne connaît que la classe abstraite `Figure`. Il y a donc encapsulation de la famille des figures par l'intermédiaire de l'interface de la classe abstraite `Figure`. Les variations dans ce cas sont modélisées par l'héritage et par l'agrégation. L'héritage est utilisé pour distinguer un rectangle d'un cercle (leurs paramètres diffèrent). L'agrégation est utilisée pour modéliser la variation dans leur comportement. Ainsi, lorsque un objet de type `Rectangle` exécute sa fonction membre `Dessine()`, il fait appel à la fonction membre `DessineLigne()` de la classe `TraçageGDI` qui est associée à la classe `Figure`. De même pour un objet de type `Cercle`. Il fera affaire avec la fonction membre `DessineCercle()` associée à sa surclasse `Figure`.

À noter que la surclasse abstraite `Figure` ne connaît que la surclasse abstraite de `TraçageGDI`.Il y a donc double encapsulation : `Client` utilise les figures à travers l'interface de `Figure` et `Figure` utilise les fonctions de traçage à travers l'interface de `Traçage`.

Nous avons créé une hiérarchie en utilisant la surclasse abstraite `Traçage`. L'idée est de montrer la flexibilité d'une telle conception et n'est pas du tout obligatoire. Pour comprendre cette flexibilité considérons que `TraçageGDI` utilise les primitives GDI (Graphical Device Interface) de Windows pour effectuer le traçage des lignes et des cercles. Plus tard, nous désirons augmenter la performance graphique du système en adoptant le système OpenGL. Il suffit alors de créer une nouvelle classe (ex : `TraçageOpenGL`) dans la hiérarchie de `Traçage` offrant les mêmes services `DessineLigne()` et `DessineCercle()` mais en utilisant les primitives de OpenGL. Évidemment, l'objet en agrégation avec `Figure` sera de type `TraçageOpenGL` et non pas de type `TraçageGDI`.

Enfin, on retrouvera l'application de ces trois stratégies dans la plupart des patrons de conception présentés dans ce chapitre. Il est donc conseillé de bien saisir l'apport de ces stratégies avant d'entreprendre l'étude des patrons de la sous-section 9.2.

Pour apprécier l'importance des patrons dans la conception logicielle, il suffit de se tourner vers d'autres disciplines du génie pour constater l'abondance des patrons.

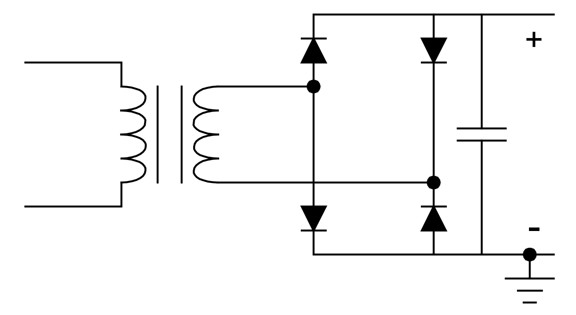


Figure 105 Redresseur double alternance monophasé.

En génie électrique, l'alimentation des appareils à courant continu passe par le redressement du courant alternatif venant du secteur mural. La plupart du temps, le secteur de 60 Hz monophasé. Le redressement consiste donc à ramener les alternances négatives en alternances positives puis lisser les ronflements résultants afin d'obtenir un courant qui est continu. Le circuit de la Figure 105 est un circuit standard réalisant cette tâche. Les praticiens l'utilise sans avoir à analyser au préalable de son comportement puisque ce circuit a été appliqué avec succès dans d'innombrables systèmes. Le circuit de la Figure 105 est un patron de conception. Il porte un nom reconnu (« Redresseur double alternance monophasé ») et son champ d'application est bien défini (« Redresse le courant alternatif en courant continu dont le secteur est monophasé »).

Un autre exemple nous provient du domaine de la mécanique hydraulique. Pour propulser un vérin hydraulique, une pompe est nécessaire pour jouer le rôle de la

source d'énergie. Si nous désirons actionner le piston en lui donnant deux vitesses possibles, le circuit de la Figure 106 est alors applicable.

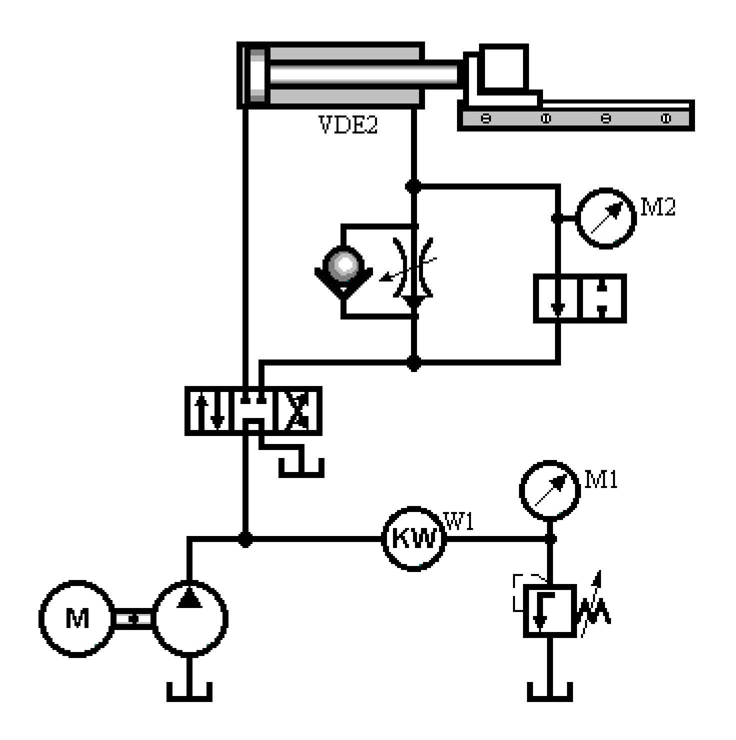


Figure 106 Circuit hydraulique à deux vitesses.

Ce circuit standard applique l'énergie hydraulique sur la prise d'entrée du vérin tandisqu'un réducteur de débit avec clapet anti-retour en parallèle avec un tiroire de court-circuit est connecté sur la prise de sortie. Un second tiroir permet l'inversion de la prise d'entrée pour la source hydraulique. Conséquemment, la vitesse de déplacement du piston peut être contrôlée par le tiroir de court-circuit. Son enclenchement à l'état ouvert laisse le réducteur de débit dans la prise de sortie du vérin procurant une vitesse de déplacement V_O. L'enclenchement à l'état fermé du tiroir élimine le réducteur de débit et donne une vitesse de déplacement V_F. Évidemment, nous avons $V_F > V_O$. Ce type de circuit est couramment utilisé par les praticiens du domaine. Le circuit hydraulique de la Figure 106 est un patron de conception. Il porte un nom reconnu (« Circuit à deux vitesses ») et son champ d'application est bien défini (« Fournir deux vitesses de déplacement au piston d'un vérin hydraulique »).

En conclusion, les patrons de conception ne sont pas propres au génie logiciel. Ils existent dans bien des disciplines et sont appliqués avec succès depuis bien longtemps. Il ne faut donc pas être réticent quant à leurs applications dans la conception des logiciels. Les patrons de conception sont présentés de la manière suivante :

- Le nom du patron (indiqué dans le titre de la sous-section);
- Le contexte d'application;

- La solution;
- Le diagramme explicatif correspondant;
- Exemple d'application.

9.2.1 PATRON « ÉTAT »

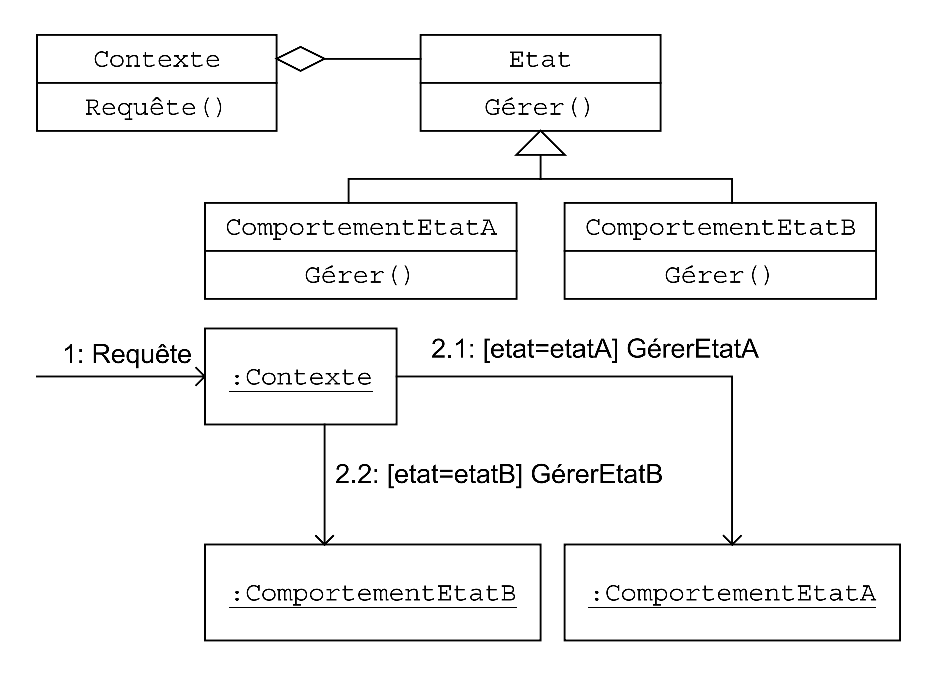
Situation :

Le comportement d'un objet est dépendant de son état. L'application de la logique conditionnelle (les `if-then-else`) est trop complexe ou n'est pas désirable.

Solution :

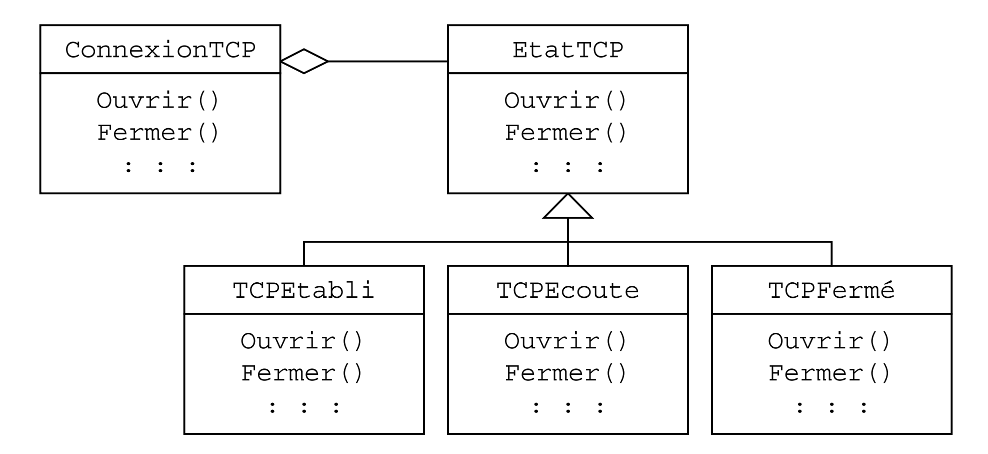
1. Créer une classe pour chaque état (objet-état) qui influence le comportement de l'objet (objet-contexte).
2. Utiliser le polymorphisme et assigner les méthodes (fonctions membres) à chacun des objets-états pour gérer le comportement de l'objet-contexte.
3. Lorsque l'objet-contexte reçoit un message qui affecte son état, le transmettre aux objets-états.

Diagramme de classes :



À la réception d'un message en provenance d'un client, l'objet-contexte délègue son comportement à l'un de ses objets-état. La délégation s'effectue selon la requête du client et le choix de l'objet-état peut être décidé par l'objet-contexte ou par les objets-états eux-mêmes.

Exemple d'application :



La classe `ConnexionTCP` maintient un objet-état qui est une instance des sous-classes de `EtatTCP` représentant l'état courant de la connexion TCP. La classe `ConnexionTCP` délègue les requêtes qui ont rapport avec son état à son objet-état. L'objet de `ConnexionTCP` changera son objet-état lorsqu'il y a changement dans l'état de la connexion TCP. Ainsi, après l'établissement de la connexion, `ConnexionTCP` replacera son objet-état par une instance de `TCPEtabli`. De même, lorsqu'il y a fermeture de la connexion, `ConnexionTCP` replacera l'instance de `TCPEtabli` par une instance de `TCPFermé`.

9.2.2 PATRON « FAÇADE »

Situation :

Un système existant possède une interface qui est complète et complexe. Nous désirons utiliser uniquement un sous-ensemble des capacités de ce système complet. Ou encore, nous désirons utiliser le système d'une façon particulière.

Solution :

1. Créer une classe qui possède l'interface requise.

2. Fait en sorte que l'interface de la classe utilise les fonctionnalités désirées du système existant.

Diagramme et exemple d'application :

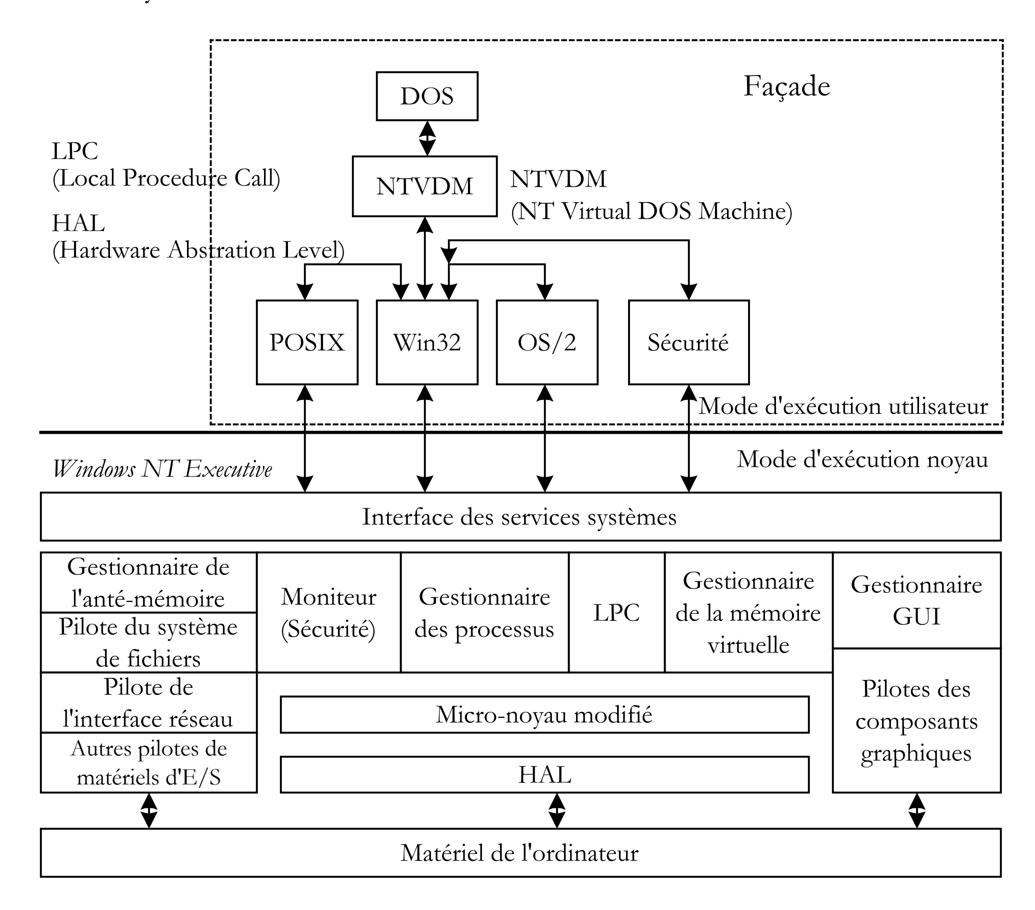
Cet exemple est extrême. Néanmoins vous pouvez appliquer ce patron à des problèmes de petite taille.

L'organisation du système d'exploitation Windows NT/2000 est un bel exemple du patron façade. Ce système d'exploitation est un complet et complexe. Il est composé d'un grand nombres de sous-systèmes interreliés. Pour simplifier son utilisation, les concepteurs de Windows NT/2000 ont créés des façades qui présentent une image consistante de l'ensemble du système. Ainsi, nous avons :

- La façade POSIX qui est compatible avec le standard Portable Operating System Interface;

- La façade Win32 pour les appels de système natifs;
- La façade OS/2 compatible avec le système d'exploitation d'IBM du même nom;
- La façade Sécurité qui est entièrement consacrée à la sécurité (système et réseau);
- La façade DOS pour la compatibilité des applications patrimoines.

Toutes ces façades reposent sur une autre façade de base appelée « Interface des services systèmes ».



9.2.3 PATRON « ADAPTATEUR » (ADAPTER PATTERN)

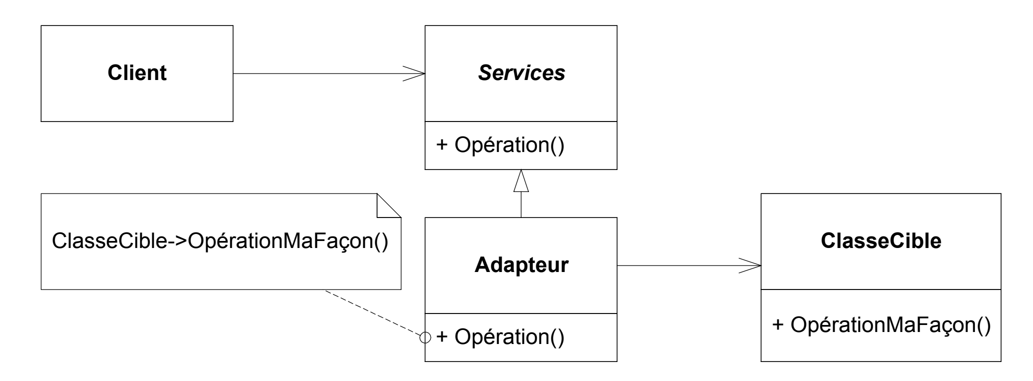
Situation :

Une classe cible contient les bonnes données et le bon comportement. Cependant, elle dispose d'une interface incompatible ou inconvénient à l'utilisation.

Solution :

1. Créer une classe adaptateur.
2. Envelopper l'interface de la classe cible par celle de la classe adaptateur.

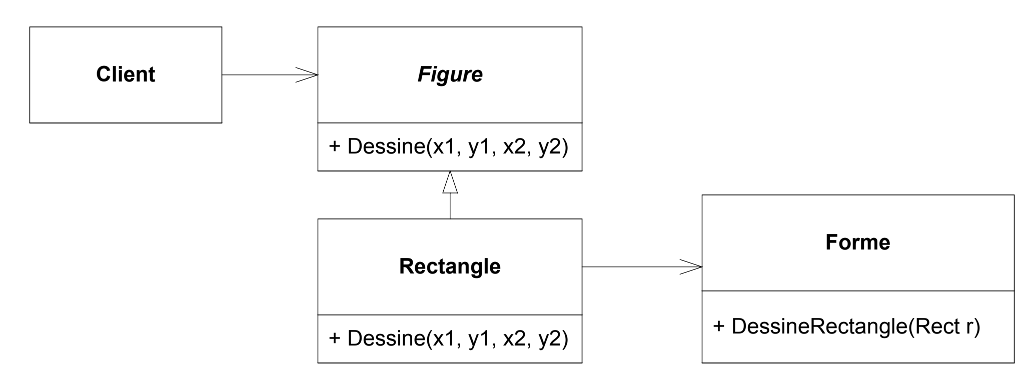
Diagramme de classe :



Le client désire obtenir le service `Opération()`. Ce service est déjà offert par `ClasseCible`. Malheureusement, son interface ne convient pas à la classe `Client`. On instaure alors une classe `Adaptateur` présentant une interface au service `Opération()` qui est convenable pour le client. À l'intérieur du service `Adaptateur::Opération()`, on traduit les spécifications du client en spécification utilisable par `ClasseCible::OpérationMaFaçon()`.

Exemple d'application :

Le client désire dessiner une figure à l'écran. Dans cet exemple, la figure est un rectangle. Nous avons en banque une classe `Forme` en mesure de dessiner un rectangle mais elle exige une structure de type `Rect` comme paramètre alors que le client s'attend à fournir les paramètres sous forme de `x1`, `y1`, `x2`, `y2`. On ne veut pas modifier le code du client pour changer toutes les occurrences de `x1`, `y1`, `x2`, `y2` en structures `Rect`. Le patron Adaptateur peut résoudre ce problème.




```
class Forme {
public :
 Forme() { }
 void DessineRectangle(Rect r) {
  // Dessiner le rectangle dont les coordonnées sont contenues dans
  // la structure r.
 }
};
class Figure { // classe abstraite
public :
 Figure() { }
```

```
virtual void Dessine(int x1, int y1, int x2, int y2) = 0;
};
// Classe Rectangle joue le rôle de l'Adapteur.
class Rectangle : public Figure { // Rectangle utilise Forme
private :
 Forme *forme;
public :
 Rectangle() {
  forme = new Forme();
 }
 // Adapter l'utilisation de DessineRectangle() pour le client
 void Dessine(int x1, int y1, int x2, int y2) {
  Rect rect;
  rect.x1 = x1; rect.x2 = x2; rect.y1 = y1; rect.y2 = y2;
  forme-> DessineRectangle(rect);
 }
};
```

9.2.4 PATRON « PONT » (BRIDGE PATTERN)

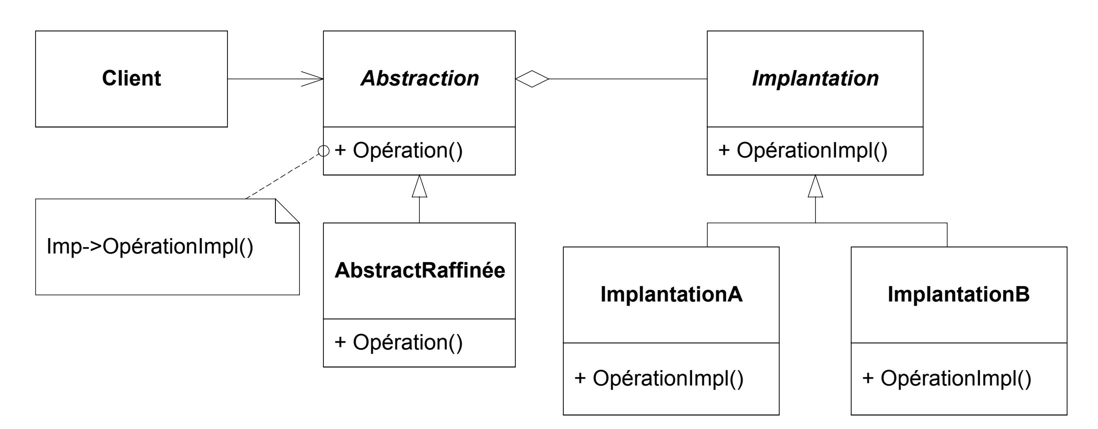
Situation :

Il est nécessaire de séparer l'abstraction (lire modèle) de son implantation. Permettre l'évolution indépendante de ces deux facettes. En d'autres mots, découpler l'ensemble des objets d'implantation de l'ensemble des objets qui les utilisent.

Solution :

1. Créer deux hiérarchies de classes.
2. Une hiérarchie représente l'abstraction (lire modèle) et le raffinement du modèle.
3. L'autre hiérarchie représente l'implantation de l'abstraction et de son raffinement.
4. Joindre les deux hiérarchies par la composition ou l'agrégation.

Diagramme de classe :

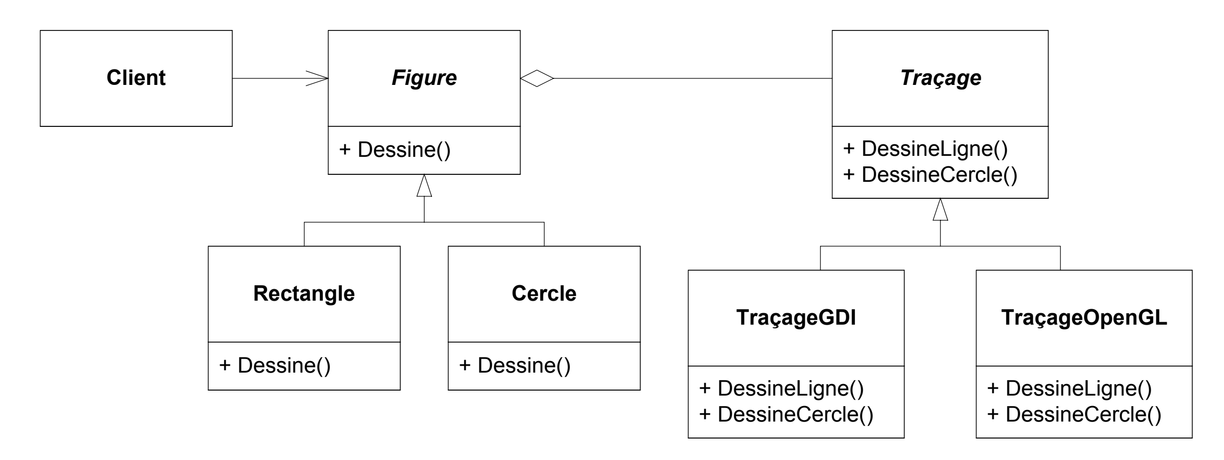


La hiérarchie de l'abstraction est séparée de la hiérarchie d'implantation. On peut

donc faire évoluer ces deux hiérarchie indépendamment. Ces deux hiérarchies sont liées par un pointeur de type `Implantation` dans la classe `Abstraction`. Le service `Opération()` de l'abstraction utilise `OpérationImpl()` de l'implantation. Ce qui permet un découplage très prononcé entre le modèle et son implantation. La composition (ou l'agrégation) liant les deux hiérarchies ressemble à un pont entre deux rives d'où le nom de ce patron.

Exemple d'application :

Encore une fois, nous avons affaire à un système de traçage. Nous avons une abstraction qui représente les figures géométriques (`Rectangle`, `Cercle`, etc.). La manifestation visuelle de ces figures est réalisée par une autre hiérarchie de classes.



Au début de la conception, le système graphique envisagé était le GDI de Windows. Par la suite, l'incorporation de OpenGL est devenue nécessaire. À cause de la séparation entre le modèle et son implantation, l'ajout de OpenGL devient une simple procédure d'héritage horizontal. Nous pouvons poursuivre le même raisonnement pour incorporer d'autres implantations (ex : DirectX).

```
// La hiérarchie d'implantation
class Tracage {
public :
 virtual void DessineLigne(int x1, int x2, int y1, int y2) = 0;
 virtual void DessineCercle(int x, int y, int rayon) = 0;
};
class TracageGDI : public Tracage { // Utilise les primitives de GDI
public :
 void DessineLigne(int x1, int x2, int y1, int y2) {
  // tracer une ligne avec une primitive GDI
 }
 void DessineCercle(int x, int y, int rayon) {
  // Tracer un cercle avec une primitive GDI
 }
};
class TracageOpenGL : public Tracage { // utilise les primitives OpenGL
public :
 void DessineLigne(int x1, int x2, int y1, int y2) {
  // tracer une ligne avec une primitive OpenGL
 }
 void DessineCercle(int x, int y, int rayon) {
  // Tracer un cercle avec une primitive OpenGL
 }
```

```
};

// La hiérarchie d'abstraction
class Figure {
protected :
 Tracage *tracage;
public :
 Figure(Tracage *tr) { tracage = tr);
 virtual void Dessine() = 0;
};
class Rectangle : public Figure {
private :
 int _x1, _x2, _y1, _y2;
public :
 Rectangle(Tracage *tr, int x1, int y1, int x2, int y2)
  : Figure(tr)
 {
  _x1 = x1; _x2 = x2; _y1 = y1; _y2 = y2;
 }
 // Implantation est responsable de son affichage visuel
 void Dessine() {
  tracage-> DessineLigne(_x1, _y1, _x2, _y1);
  tracage-> DessineLigne(_x2, _y1, _x2, _y2);
  tracage-> DessineLigne(_x2, _y2, _x1, _y2);
  tracage-> DessineLigne(_x1, _y2, _x1, _y1);
 }
};
class Cercle : public Figure {
private :
 int _x, _y, _r;
public :
 Cercle(Tracage *tr, int x, int y, int rayon)
  : Figure(tr)
 {
  _x = x; _y = y; _r = rayon;
 }
 // Implantation est responsable de son affichage visuel
 void Dessine() {
  tracage->DessineCercle(_x, _y, _r);
 }
};
```

9.2.5 PATRON « MANUFACTURE ABSTRAITE » (ABSTRACT FACTORY PATTERN)

Situation :

Un ensemble ou une famille d'objets sont à créer. On ne veut pas utiliser la construction switch – case pour réaliser cette tâche. Parce qu'une longue liste de switch – case n'est pas très *orientée objet* et la logique de création des objets (paramètres initiaux, dépendance, etc.) est difficile à maintenir (ou à modifier).

Solution :

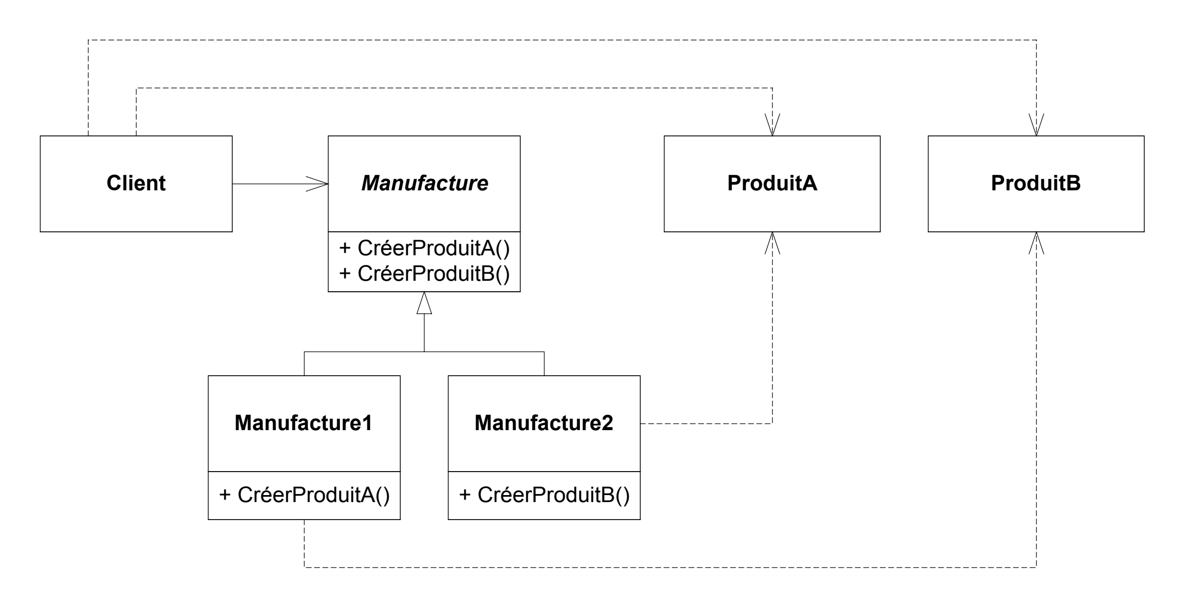
1. Créer une **classe manufacture** par famille d'objets à créer. Donc, autant de classes manufactures que de familles d'objets.

2. Doter cette classe manufacture la logique et l'interface nécessaires pour créer les objets de la même famille.

3. Le client crée les objets désirés par l'intermédiaire des classes manufactures.

4. Le client ne doit pas créer les objets désirés directement.

Diagramme de classes :



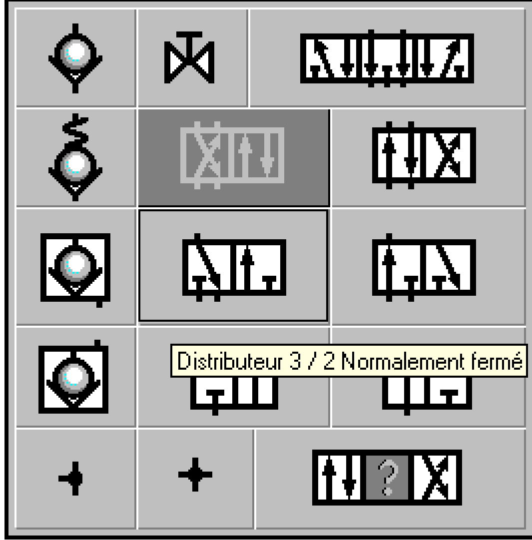
Le client désire obtenir un objet de type `ProduitA`. Il passe la commande en exécutant le service `CréerProduitA`. Dans ce cas, la `Manufacture1::CréerProduitA()` contient la logique nécessaire pour créer un objet de type `ProduitA`. On peut utiliser la même démarche logique pour la création d'un objet de type `ProduitB`. De même, on peut étendre la logique à plus d'un type d'objet par famille (ex : `ProduitA1`, `ProduitA2`, `ProduitA3`, etc.). Ainsi, la logique de création (paramètres initiaux, dépendance, etc.) des objets est très bien encapsulée.

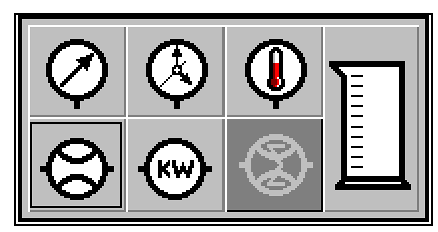
Exemple d'application :

Un simulateur de systèmes hydrauliques et pneumatiques comprend une gamme de composants et d'appareils de mesure. Ces composants hydrauliques et pneumatiques sont disposés dans des palettes de composants. L'utilisateur peut les sélectionner et les déposer sur la feuille de travail du simulateur.

Dans le simulateur, la création de ces composants est la responsabilité de l'objet `Éditeur`. Ce dernier est capable de détecter le déplacement et l'état des boutons de la souris. Après un clic de la souris sur la case d'un composant, l'Éditeur est en mesure de déterminer le type et la famille du composant sélectionné par l'utilisateur (à l'aide des coordonnées de la souris). L'`Éditeur` passe alors la commande à la manufacture appropriée pour réaliser la création de l'objet désiré. Ainsi, la logique de création est déplacée de l'`Éditeur` rendant plus simple le maintien de cet objet. Par contre, l'`Éditeur` demeure le responsable principal de la création des objets, ce qui est tout à fait logique.








```
// La hiérarchie des manufactures
class Manufacture {
public :
 virtual void CreerParType (int type) = 0;
 // Famille des pompes
 void CreerPompe () { }
 void CreerPompeCV() { }
 : : :
 // Famille des distributeurs
 void CreerDistOuvertFerme() { }
 void CreerDist32() { }
 : : :
 // Famille des appareils de mesure
 void CreerManometre() { }
 void CreerWattmetre() { }
 : : :
};
class ManufacturePompe : public Manufacture {
public :
 void CreerParType (int type) { // créer le composant par son type }
 // Famille des pompes
 void CreerPompe () { // créer une pompe }
 void CreerPompeCV() { // créer une pompe à cylindrée variable }
 : : :
};
class ManufactureDistributeur : public Manufacture {
public :
 void CreerParType (int type) { // créer le composant par son type }
 // Famille des distributeurs
 void CreerDistOuvertFerme () { // créer distributeur ouvert-fermé }
 void CreerDist32() { // créer distributeur 3-2 }
 : : :
};
class ManufactureAppMesure : public Manufacture {
public :
 void CreerParType (int type) { // créer le composant par son type }
 // Famille des distributeurs
```

```
 void CreerManometre() { // créer un manomètre }
 void CreerWattmetre() { // créer un Wattmètre }
 : : :
};
// supposons que les classes représentant les composants sont définies
// ailleurs
// L'éditeur responsable principal de la création des composants
class Editeur {
private :
 Manufacture *manufacture;
// Nous montrons seulement le code relatif à la création des composants
public :
 // Voyez le switch – case est maintenant très simple à gérer
 CréerComposant(FAMILLE famille, UINT type) {
  switch (famille) {
  case POMPE :
    manufacture = new ManufacturePompe();
    manufacture-> CreerParType (type);
    delete manufacture;
    break;
  case DISTRIBUTEUR :
    manufacture = new ManufactureDistributeur();
    manufacture-> CreerParType (type);
    delete manufacture;
    break;
  case APPAREIL_MESURE :
    manufacture = new ManufactureAppMesure();
    manufacture-> CreerParType (type);
    delete manufacture;
    break;
  }
 }
};
```

9.2.6 PATRON « STRATÉGIE » (STRATEGY PATTERN)

Situation :

Différents contextes demandent différentes approches pour résoudre un problème. Nous avons une famille d'algorithmes mais ils ne sont applicables que dans des contextes bien précis. La sélection de l'algorithme dépend de la requête du client.

Solution :

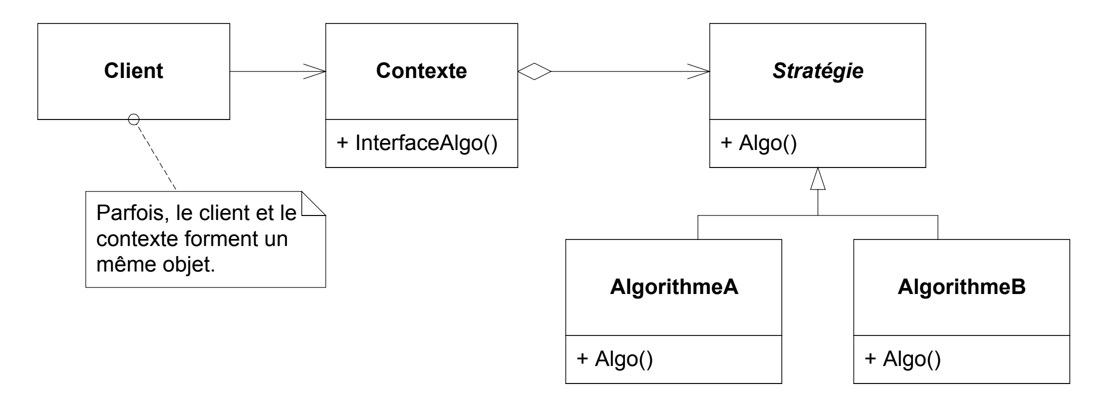
1. Créer une classe abstraite de stratégie. Donner à cette classe une interface uniforme qui est applicable à tous les algorithmes encapsulés.

2. Créer une classe par algorithme en héritant de la classe abstraite de stratégie.

3. Créer une classe de contexte (une sorte de sélecteur d'algorithmes) et lier cette classe à celle de stratégie par composition ou agrégation.

4. Dans ce patron, l'invocation des algorithmes est réalisée de la même façon pour tous les algorithmes encapsulés.

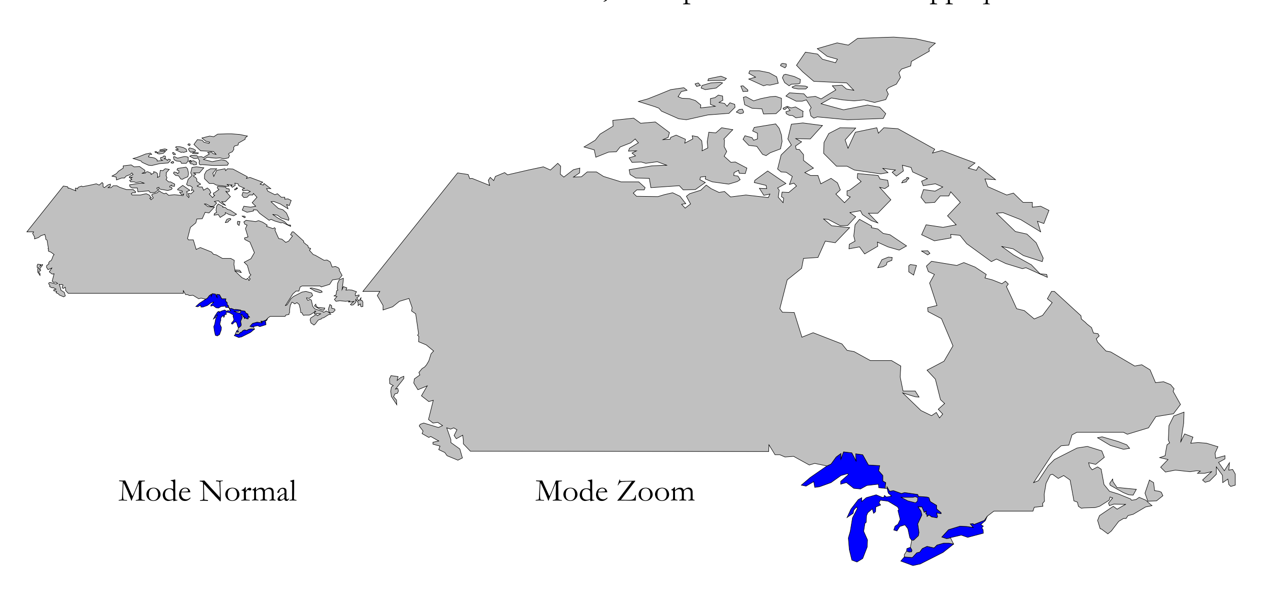
Diagramme de classes :



La classe `Contexte` contient un objet de type `Stratégie`. La classe Stratégie, quant à elle, contient un service `Algo()` qui agit comme interface aux algorithmes concrets.Chacune des classes dérivées réalise l'implantation d'un algorithme. Le client indique le contexte d'exécution de l'algorithme et c'est la responsabilité de l'objet `Contexte` de déterminer l'algorithme (`AlgorithmeA` ou `AlgorithmeB`) à utiliser.

Exemple d'application :

Un système de cartographie informatisée dispose d'une fonction de visualisation « Zoom ». En mode de visualisation normale, la carte est montrée à l'écran selon une échelle de 500 Km par cm. L'option « Zoom » modifie cette échelle à 100 Km par cm. Ainsi, le contour des routes doit être plus précis en mode « Zoom ». La méthode d'interpolation non linéaire est utilisée pour tracer le contour des routes en mode zoom. Malheureusement, l'interpolation non linéaire est très longue à calculer surtout le remplissage des régions qui exige beaucoup de tests. Conséquemment, en mode normal de visualisation, l'interpolation linéaire est appliquée.



```
// La famille des algorithmes
class Strategy {
public:
 virtual void Algo(Donnees d) = 0;
};
class InterpolationLineaire : Strategy {
public:
 void Algo(Donnees d) { // exécuter l'interpolation linéaire sur d }
};
class InterpolationNonLineaire : Strategy {
public:
 void Algo(Donnees d) { // exécuter l'interpolation non linéaire sur d }
};
// Le contexte d'application des algorithmes
class Contexte {
private:
 Strategy *s;
public:
 // Exécute la bonne méthode en fonction de la variable Zoom
 void ExecuteInterpolation(Donnes data, bool Zoom) {
  if (Zoom) {
   s = new InterpolationNonLineaire();
  } else {
   s = new InterpolationLineaire();
  }
  s->Algo(data); // exécuter le bon algorithme
  delete s;
 }
};
```

9.2.7 PATRON « SINGLETON »

Situation :

La classe `ClasseA` n'a qu'une seule instance (On ne peut que créer un seul objet de type `ClasseA`). Un objet global responsable pour la création n'est pas désiré. De plus, on veut un mécanisme automatique qui force les programmeurs à ne créer qu'une seule instance de ce type.

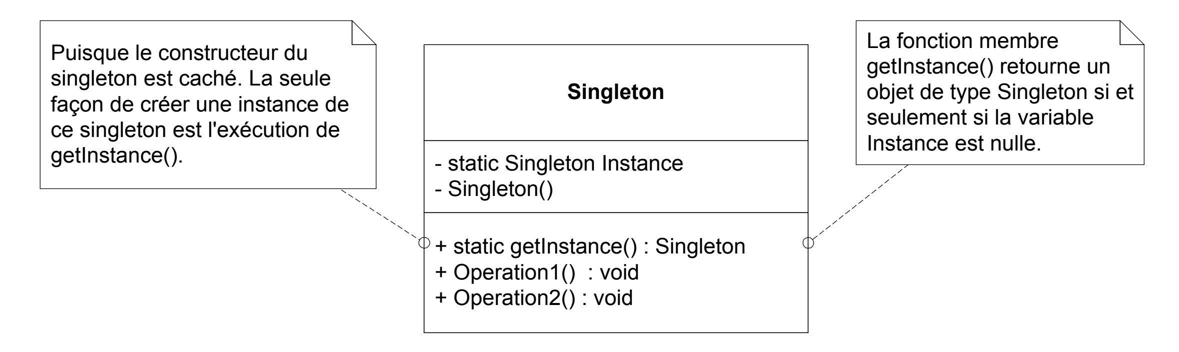
Solution :

1. Instaurer une variable statique dans la classe. Cette variable statique servira à indiquer l'existence d'une instance de la classe.

2. Cacher le constructeur de la classe dans la section protégée ou privée de la classe.

3. Instaurer une fonction membre statique pour la création de l'objet. Cette fonction membre exécute sa tâche seulement si la variable statique d'instance est nulle.

Diagramme de classes :



Exemple d'application :

Dans un simulateur de systèmes hydrauliques et pneumatiques, il existe un et un seul Éditeur par session de travail.

Hydro+Pneu - Sans Nom

Fichier Edition Simulation Aide

Abadi MT Condensed Light 12

Un seul éditeur par session de travail

ISO Français Hydraulique CAPS NUM SCRL OVR

De toute évidence ce simulateur est réalisé sous forme d'une application SDI (*Single Document Interface*). Ainsi, la classe `Éditeur` de ce simulateur peut être conçu comme une classe singleton.

```
// Editeur est un singleton
Editeur *Editeur::Instance = 0;

class Editeur {
private:
 static Editeur Instance;
 // Cacher son constructeur
 Editeur() { }
 // Autres données et fonctions de l'éditeur
 : : :
public:
 // getInstanceEditeur s'opère de la manière suivante :
 // 1) Aucun objet de type Editeur n'est encore créé alors créer un objet
 // de ce type et retourner l'objet.
 // 2) Un objet de type Editeur existe déjà alors retourner l'objet déjà créé.
 static Editeur *getInstanceEditeur() {
  if (Instance == 0)
   Instance = new Editeur();
  return Instance;
 }
 // Autres données et fonctions de l'éditeur
 : : :
};


// Voici comment on peut créer correctement l'Éditeur
Editeur *ed1 = Editeur::getInstanceEditeur();
// Un deuxième appel à getInstanceEditeur() retournera le même objet.
// C'est à dire ed1 == ed2
Editeur *ed2 = Editeur::getInstanceEditeur();
// Ici le compilateur affichera un message d'erreur
Editeur *Mauvais1 = new Editeur;     // Erreur !
// de même qu'ici …
Editeur Mauvais2;    // Erreur !
```

9.2.8 PATRON « OBSERVATEUR » (OBSERVER PATTERN)

Situation :

Un événement survient et nous devons notifier un ensemble d'objets de l'arrivée de cet événement. Pour certains des objets, ils doivent également changer d'état en fonction de l'événement reçu.

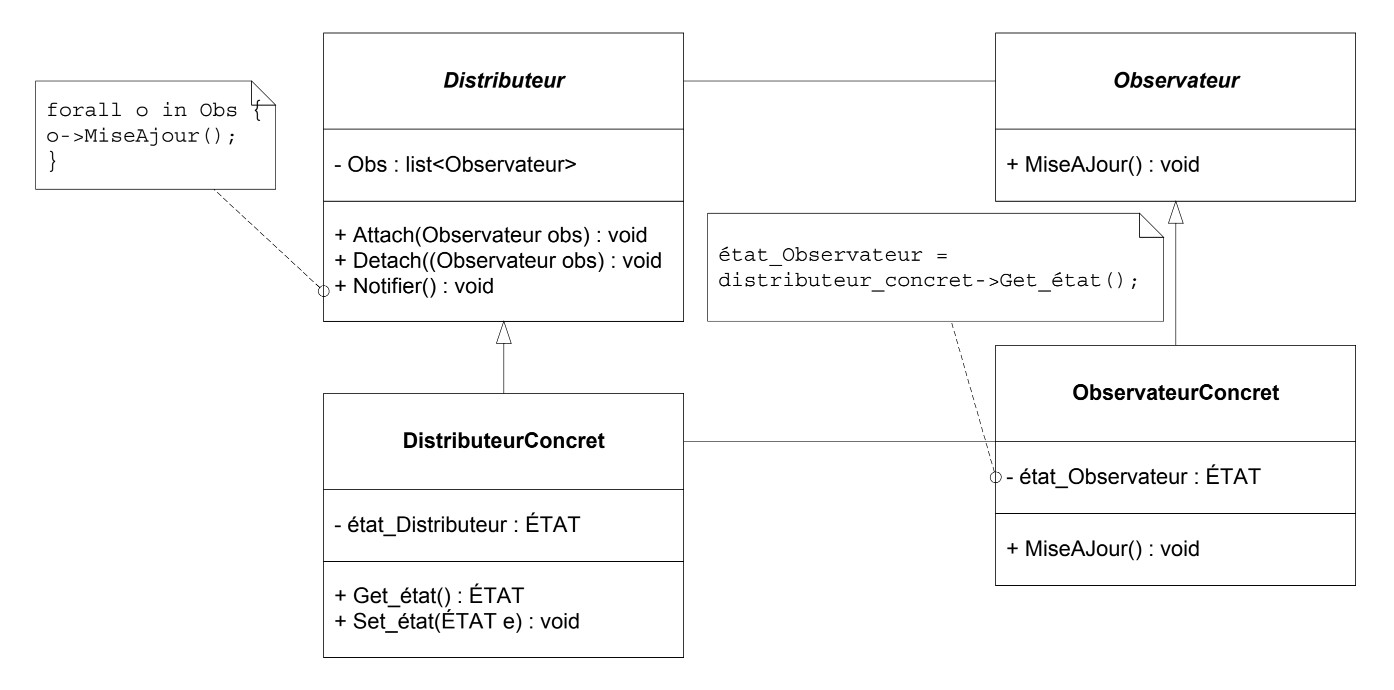
Solution :

1. Créer une classe `Distributeur` qui est au courant des événements dans le système.

2. Chaque objet désireux de recevoir la notification des événements reçus doit s'enregistrer auprès d'un objet de type `Distributeur`. Les objets qui reçoivent la notification des événements sont des *observateurs*.

3. Les observateurs peuvent également engendrer des événements. Ces événements pourront être distribués aux autres observateurs par le biais de l'objet distributeur.

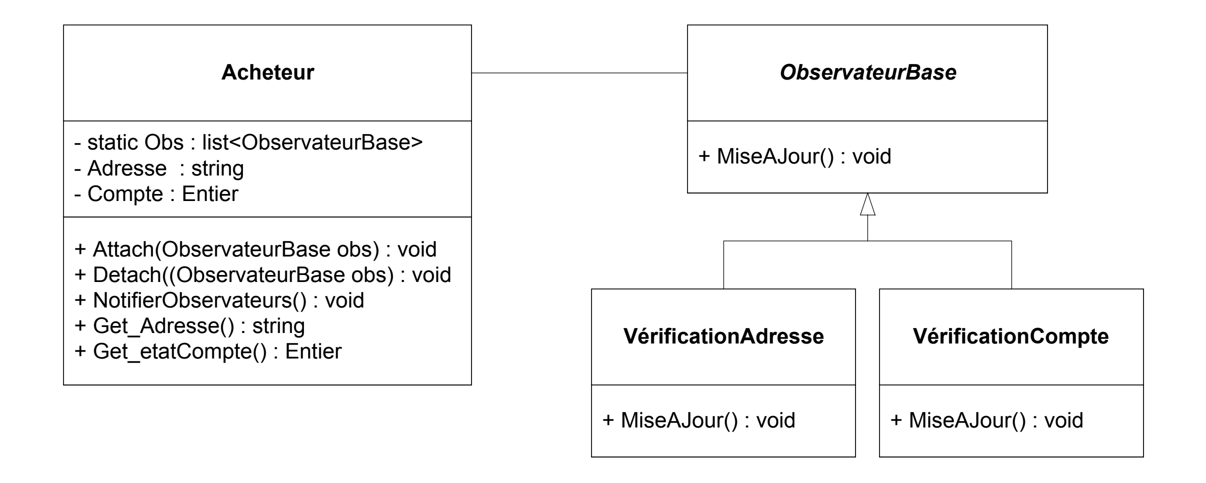
Diagramme de classes :



Un objet `Observateur` doit s'enregistrer auprès de l'objet `Distributeur`. à l'aide de la fonction membre `Distributeur::Attach()`. Une fois attaché, l'objet `Observateur` peut recevoir la notification de l'arrivée des événements. Plus d'un observateur peut ainsi s'enregistrer auprès du distributeur. La notification s'effectue en parcourant la liste `Obs` et en exécutant la fonction membre `Observateur::MiseAJour()`. Le changement d'état des observateurs est normalement réalisé en considérant la nature de l'événement reçu. Dans ce diagramme de classes, la nature de l'événement reçu est identifiée par l'attribut `état_Distributeur`. La valeur de `état_Distributeur` est propagée à l'attribut `état_Observateur` à l'aide de la fonction membre `DistributeurConcret::Get_état()`.

Exemple d'application :

Dans un système de vente informatisé, chaque client acheteur entré dans le système est automatiquement validé à l'aide de son adresse et de son état de compte. Le patron Observateur peut aider dans une telle situation. Les observateurs concrets dans ce cas sont les classes `VerificationAdresse` et `VerificationCompte`. Le distributeur est représenté par la classe `Acheteur`. Les informations nécessaires à la validation sont contenues dans la classe `Acheteur` elle-même.




```
// déclaration anticipée
class ObservateurBase;
// Acheteur est le distributeur
class Acheteur {
private:
 static vector< ObservateurBase> Obs;
 void NotifierObservateurs() {
  vector< ObservateurBase>::iterator i;
  for (i = Obs.begin(); i!=Obs.end(); i++)
   i->MiseAJour(this);
 }
public:
 static void attach(ObservateurBase *o);
 static void Detach(ObservateurBase *o);
 string Get_Adresse();       // retourne l'adresse de l'acheteur
 int Get_etatCompte();       // retourne l'état de compte de l'acheteur
};

class ObservateurBase {
public:
 ObservateurBase () {
  // s'enregistrer auprès du distributeur
  Acheteur.attach(this);
 }
 void MiseAJour(Acheteur *a) = 0;
};

// VerificationAdresse est un observateur concret
class VerificationAdresse : public ObservateurBase {
public:
 VerificationAdresse() { }
 void MiseAJour(Acheteur *a) {
  // valider l'adresse de l'acheteur. Cette information peut être
  // obtenue via a-> Get_Adresse()
 }
};
// VerificationCompte est aussi un observateur concret
class VerificationCompte : public ObservateurBase {
public:
 VerificationCompte () { }
 void MiseAJour(Acheteur *a) {
  // vérifer l'état de compte de l'acheteur. Cette information peut être
  // obtenue via a->Get_etatCompte()
 }
};
```

9.2.9 PATRON « DÉCORATEUR » (DECORATOR PATTERN)

Situation :

Un objet dispose d'un nombre de fonctions. Il est souhaité de pouvoir ajouter de nouvelles fonctionnalités à l'objet et ce, dynamiquement pendant l'exécution du programme.

Solution :

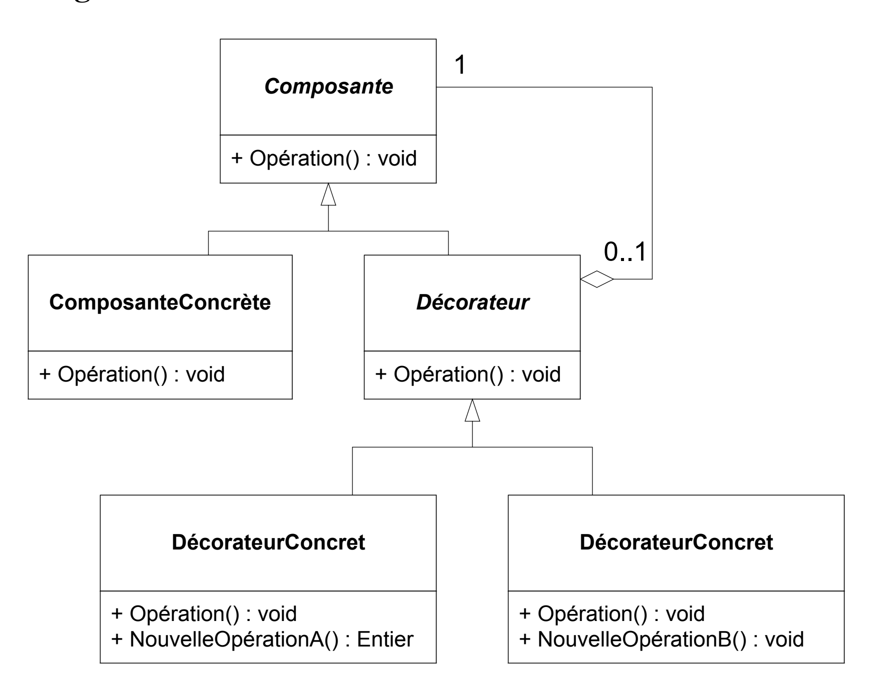
1. Créer une classe abstraite qui représente à la fois la classe originale et les nouvelles fonctions à ajouter.

2. La classe originale (celle où l'on désire ajouter de nouvelles fonctionnalités) est appelée la *composante abstraite*.

3. Les classes renfermant les nouvelles fonctions sont appelées les *décorateurs*.

4. Instancier les objets appropriés à partir de la composante abstraite.

Diagramme de classes :

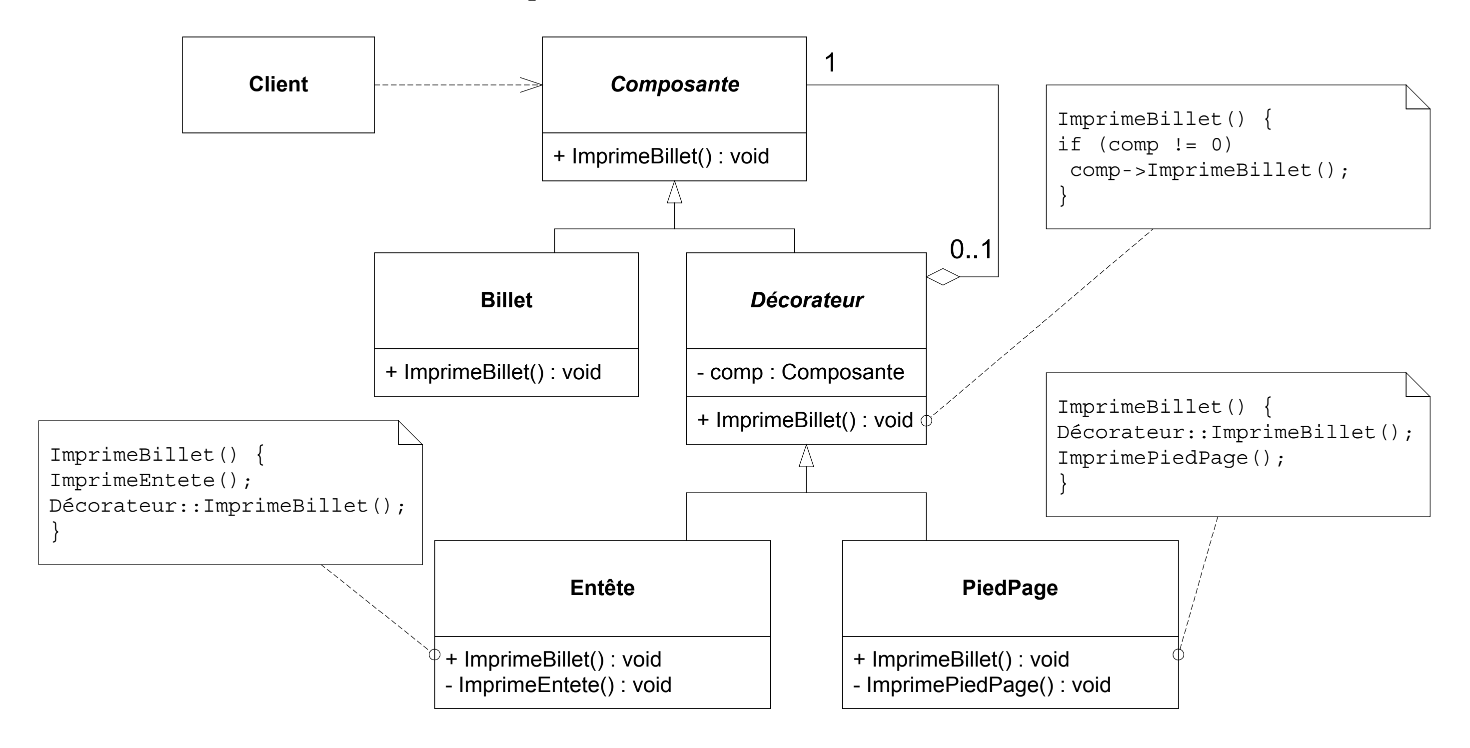


La composante abstraite (`Compoasnte`) peut contenir un décorateur abstrait (`Décorateur`). La cardinalité de cette composition ou l'agrégation est de 0 ou 1. Donc, la fonctionnalité de la composante n'est augmentée que si elle contient un décorateur.

Puisque la composante concrète hérite de la composante abstraite, elle peut disposer de nouvelle fonctionnalité dynamiquement lors l'exécution du programme.

Exemple d'application :

Un système de billetterie imprime ses billets avec, parfois, un en-tête annonçant une nouvelle option pour les acheteurs et, d'autrefois, un pied de page indiquant le commanditaire de l'événement (s'il existe). On peut réaliser aisément ces variations en utilisant le patron Décorateur.



Dans la composante concrète `Billet`, La fonction membre `ImprimeBillet()` imprimera le corps du billet. La fonction membre du décorateur `Entête`, quant à elle, imprimera l'en-tête du billet et ensuite demandera à sa classe de base d'imprimer à son tour. De même pour le décorateur `PiedPage`. Il demandera à sa classe de base d'imprimer d'abord puis il imprimera le texte du pied de page.

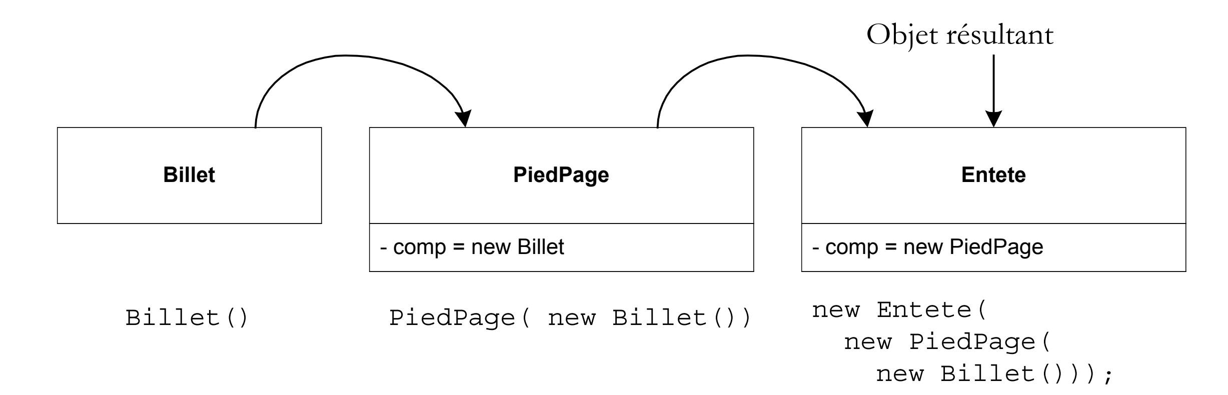
La classe de base des décorateurs est le décorateur abstrait. Dans la fonction membre `Decorateur::ImprimeBillet()`, un test est effectué pour connaître l'état de la variable `comp`. Si `comp != 0` alors on exécute sa fonction membre `ImprimeBillet()`. Dans le cas contraire `Decorateur::ImprimeBillet()` ne fera rien.

On peut donc réaliser un **chaînage des objets** de la manière suivante :

1. Créer les objets dans l'ordre désiré. Par exemple :

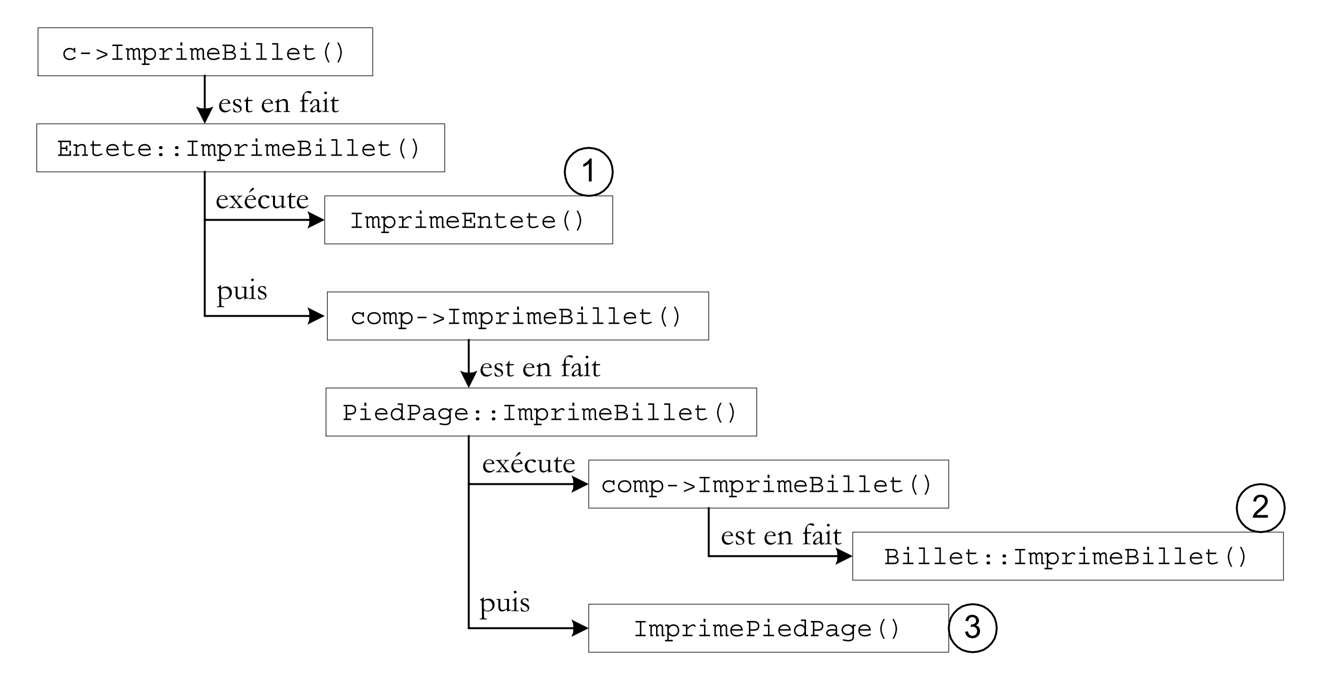
```
Composante *c = new Entete( new PiedPage( new Billet()));
```

La figure ci-dessous montre l'effet de ce chaînage :



Le résultat est un objet de type `Entete` contenant un objet de type `PiedPage` contenant un objet de type `Billet`. Ainsi le pointeur `Composante *c` pointera vers un objet de type `Entete`.

2. Exécuter `c->ImprimeBillet()`. Voici les activités résultantes de ce chaînage d'objets :



- `c->ImrimeBillet()` est en fait `Entete::ImprimeBillet()`. Dans `Entete::ImprimeBillet()`, on exécute d'abord `ImprimeEntete()` puis `comp->ImprimeBillet()`.

- À cause du chaînage `comp->ImprimeBillet()` est en fait `PiedPage::ImprimeBillet()`. Dans `PiedPage::ImprimeBillet()`, on exécute d'abord `comp->ImprimeBillet()` puis `ImprimePiedPage()`.

- À cause du chaînage `comp->ImprimeBillet()` est en fait `Billet::ImprimeBillet()`.

```
// La classe Composante abstraite n'est pas montrée
//
// Classe Composante concrète
class Billet : public Composante {
public:
 void ImprimeBillet() { // imprimer le billet }
};

// Classe Décorateur abstrait
class Decorateur : public Composante {
private:
 Composante *comp;
public:
 Decorateur(Composante *c) { comp = c; }
 virtual void ImprimeBillet() {
  if (comp != 0)
   comp->ImprimeBillet();
 }
};

// Classe Entete est un décorateur concret
class Entete : public Decorateur {
private:
 void ImprimeEntete() { // Imprime le texte de l'entête }
public:
 void ImprimeBillet() {
  // 1) Imprimer Entete
  ImprimeEntete();
  // 2) Imprimer le billet
  Decorateur::ImprimeBillet();
 }
};

// Classe PiedPage est un décorateur concret
class PiedPage : public Decorateur {
private:
 void ImprimePiedPage() { // Imprime le texte pied de page }
public:
 void ImprimeBillet() {
  // 1) Imprimer le billet
  Decorateur::ImprimeBillet();
  // 2) Imprimer le pied de page ici
  ImprimePiedPage();

 }
};

// Voici comment on peut utiliser la composante concrète
Composante *c;
c = new Entete( new PiedPage( new Billet())); // Wow !!
c->ImprimeBillet();
```

LECTURE SUGGÉRÉE

Les références qui ont aidé à la rédaction de ce chapitre sont :

[SHAL02] Shalloway, A., Trott, J., *Design Patterns Explained*, Addison-Wesley, 2002.

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns : Element of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[MART95] Martin, J., Odell, J., *Object-Oriented Methods : A Foundation*, Prentice-Hall, 1995.

[LARM98] Larman, Craig, *Applying UML and Patterns. An introduction to object-oriented analysis and design*, Prentice-Hall, 1998.

[PRES00] Pressman, R.S., *Software Engineering : A practitionner's approach*, McGraw-Hill, 2000.

[MULL97] Muller, Pierre-Alain, *Instant UML*, Wrox Press, 1997.

PROBLÈMES

À venir